Nouns DAO Security Review

Auditors

Rajeev, Lead Security Researcher

hyh, Lead Security Researcher

r0bert, Security Researcher

Christos Papakonstantinou, Junior Security Researcher

tchkvsky, Junior Security Researcher

Report prepared by: Christos Papakonstantinou and tchkvsky

July 19, 2023

Contents

1 About Spearbit

Spearbit is a decentralized network of expert security engineers offering reviews and other security related services to Web3 projects with the goal of creating a stronger ecosystem. Our network has experience on every part of the blockchain technology stack, including but not limited to protocol design, smart contracts and the Solidity compiler. Spearbit brings in untapped security talent by enabling expert freelance auditors seeking flexibility to work on interesting projects together.

Learn more about us at spearbit.com

2 Introduction

Nouns DAO is the main governing body of the Nouns ecosystem. Nouns are generative non-fungible tokens on the Ethereum blockchain, which are 32x32 pixel characters based on people, places, and things.

One noun is generated every day (24 hours). Each noun is an irrevocable member of Nouns DAO and entitled to one vote in all governance matters.

Disclaimer: This security review does not guarantee against a hack. It is a snapshot in time of Nouns DAO according to the specific commit. Any modifications to the code will require a new security review.

3 Risk classification

Severity level	**Impact: High**	**Impact: Medium**	**Impact: Low**
Likelihood: high	Critical	High	Medium
Likelihood: medium	High	Medium	Low
Likelihood: low	Medium	Low	Low

3.1 Impact

- High - leads to a loss of a significant portion (>10%) of assets in the protocol, or significant harm to a majority of users.
- Medium - global losses <10% or losses to only a subset of users, but still unacceptable.
- Low - losses will be annoying but bearable--applies to things like griefing attacks that can be easily repaired or even gas inefficiencies.

3.2 Likelihood

- High - almost certain to happen, easy to perform, or not easy but highly incentivized
- Medium - only conditionally possible or incentivized, but still relatively likely
- Low - requires stars to align, or little-to-no incentive

3.3 Action required for severity levels

- Critical - Must fix as soon as possible (if already deployed)
- High - Must fix (before deployment if not already deployed)
- Medium - Should fix
- Low - Could fix

4 Executive Summary

Over the course of 13 days in total, Nouns DAO engaged with Spearbit to review the nouns-dao-v3 protocol. In this period of time a total of **67** issues were found.

Summary

Project Name	Nouns DAO
Repository	nouns-dao-v3
Commit	9a6f06...eda79c
Type of Project	Governance, NFT
Audit Timeline	June 1 - June 19
Two week fix period	June 19 - July 3

Issues Found

Severity	Count	Fixed	Acknowledged
Critical Risk	0	0	0
High Risk	2	1	1
Medium Risk	18	13	5
Low Risk	22	3	19
Gas Optimizations	3	1	2
Informational	22	10	12
Total	**67**	**28**	**39**

5 Findings

5.1 High Risk

5.1.1 Any signer can cancel a pending/active proposal to grief the proposal process

Severity: High Risk

Context: NounsDAOV3Proposals.sol#L560-L589, NounsDAOV3Proposals.sol#L813-L821

Description: Any proposal signer, besides the proposer, can cancel the proposal later irrespective of the number of votes they contributed earlier towards the threshold. The signer could even have zero votes because `getPriorVotes(signer,..)` is not checked for a non-zero value in `verifySignersCanBackThisProposalAndCountTheirVotes()` as part of the `proposeBySigs()` flow.

This seems to be a limitation of the design described in hackmd.io/@el4d/nouns-dao-v3-spec#Cancel. With the signature-based scheme, every signer is as powerful as the proposer. As long as their combined votes meets threshold, it does not matter who contributed how much to the voting power. And assuming everyone contributed some non-zero power, they are all given the cancellation capability. However, for example, a signer/proposer with 0 voting power is treated on par with any other signer who contributed 10 Nouns towards meeting the proposal threshold. A malicious signer can sign-off on every valid proposal to later cancel it. The vulnerability arises from a lack of voting power check on signer and the cancel capability given to any signer.

Example scenario: Evil, without having to own a single Noun, creates a valid signature to back every signature-based proposal from a different account (to bypass `checkNoActiveProp()`) and gets it included in the proposal creation process via `proposeBySigs()`. Evil then cancels every such proposal at will, i.e. no signature-based proposal that Evil manages to get included into, potentially all of them, will ever get executed.

Impact: This allows a malicious griefing signer who could really be anyone without having to own any Nouns but manages to get their signature included in the `proposeBySigs()` to cancel that proposal later. This effectively gives anyone a veto power on all signature-based proposals.

High likelihood + Medium impact = High severity.

- Likelihood is High because anyone with no special ownership (of Nouns) or special roles in the protocol could initiate a signature to be accepted by the proposer. We assume no other checks by e.g. frontend because those are out-of-scope, not specified/documented, depend on the implementation, depend on their trust/threat models or may be bypassed with protocol actors interacting directly with the contracts. We cannot be sure of *how* the proposer decides on which signatures to include and what checks are actually made, because that is done offchain. Without that information, we are assuming that proposer includes all signatures they receive.
- Impact is Medium because, with the Likelihood rationale, anyone can get their signature included to later cancel a signature-backed proposal, which in the worst case (again without additional checks/logic) gives anyone a veto power on all signature-based proposals to potentially bring governance to a standstill if signatures are expected to be the dominant approach forward. Even if we assume that a proposer learns to exclude a zero-vote cancelling signer (with external checks) after experiencing this griefing, the signer can move on to other unsuspecting proposers. Given that this is one of the key features of V3 UX, we reason that this permissionless griefing DoS on governance to be at Medium impact.

While the cancellation capability is indeed specified as the intended design, we reason that this is a risky feature for the reasons explained above. This should ideally be determined based only on the contributing voting power as suggested in our recommendation. Filtering out signers with zero voting power raises the bar from the current situation in requiring signers to have non-zero voting power (i.e. cost of griefing attack becomes non-zero) but will not prevent signers from transferring their voting power granting Noun(s) to other addresses, get new valid signatures included on other signature-based proposals and grief them later by cancelling. Equating a non-zero voting power to a veto power on all signature-based proposals in the protocol continues to be very risky.

Recommendation: Consider the following mitigative redesign (as a suggestive example i.e. there could be others) to allow cancellations only based on cumulative voting power:

1. Remove special consideration for proposer in the signature-based flow in `verifySignersCanBackThisProposalAndCountTheirVotes()` and other related logic to instead make them include their signature along with other signers.
2. Remove the logic of `msgSenderIsProposer` in `cancel()` to allow cancellations only if the cumulative votes of signers is less than `proposal.proposalThreshold`.
3. Add `cancelledSigs` like logic in cumulative voting power calculation of `cancel()` to have signers rely on it to remove their support even beyond the proposal creation state.
4. Allow anyone to permissionlessly (i.e. doesn't have to be proposer or signer) call `cancel()` which calculates the cumulative voting power calculation of the proposal while accounting for `cancelledSigs` like logic from all signers (even the proposer who is also a signatory like other signers) and cancels the proposal transactions if the cumulative voting power is below `proposalThreshold`.

The scale of the modifications proposed isn't too substantial, the base recommendation can be written as (skipping the (1) step above to limit the changes and to remain in line with V1 and V2 logic):

- NounsDAOV3Proposals.sol#L560-L587

```
    function cancel(NounsDAOStorageV3.StorageV3 storage ds, uint256 proposalId) external {
        ...

        NounsDAOStorageV3.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);
+           if (!ds.cancelledSigProposals[signers[i]][proposalId]) votes +=
↪   nouns.getPriorVotes(signers[i], block.number - 1);
        }

        require(
-           msgSenderIsProposer || votes <= proposal.proposalThreshold,
+           proposer == msg.sender || votes <= proposal.proposalThreshold,
            'NounsDAO::cancel: proposer above threshold'
        );
```

- NounsDAOInterfaces.sol#L684-L685)

```
        /// @notice user => sig => isCancelled: signatures that have been cancelled by the signer and
        ↪  are no longer valid
        mapping(address => mapping(bytes32 => bool)) cancelledSigs;
+
+       /// @notice user => proposalId => isCancelled: proposals for which signers has cancelled their
↪   signatures
+       mapping(address => mapping(uint256 => bool)) cancelledSigProposals;
```

- NounsDAOV3Proposals.sol#L262-L267

```
-   function cancelSig(NounsDAOStorageV3.StorageV3 storage ds, bytes calldata sig) external {
+   function cancelSig(NounsDAOStorageV3.StorageV3 storage ds, bytes calldata sig, uint256 proposalId)
↪   external {
        bytes32 sigHash = keccak256(sig);
        ds.cancelledSigs[msg.sender][sigHash] = true;

+       if (proposalId > 0) {
+           ds.cancelledSigProposals[msg.sender][proposalId] = true;
+       }
        emit SignatureCancelled(msg.sender, sig);
    }
```

Nouns:

> Signers being able to cancel a proposer they signed is a conscious design choice. We plan to filter out signers that have voting power of zero in `proposeBySigs`. We implemented a fix here: PR 713 but still think the severity should be lowered.
>
> Likelihood: low Only the proposer of a proposal can call `proposerBySigs`, therefore, a malicious signer can't include the signature by front-running for example. In addition, the frontend will filter out signers that have no voting power as well as notify the user that any signer included will have the power to cancel the proposal. Therefore, we think the likelihood a user would add a signer with no voting power is low
>
> Impact: low Even if for some reason, perhaps confusion, a user adds a signer with zero votes and their proposal gets cancelled, they can propose again and are unlikely to make the same mistake again.
>
> For the reasons we shared earlier: won't fix

Spearbit: The proposed fix PR 713 checks for signer votes and skips inclusion/consideration if its votes is zero. It also reverts if the number of non-zero signers is zero.

The fix for the described surface and our recommendation would be to remove the ability of a signer to cancel unconditionally. Simultaneously, in order to keep the relevant degree of renouncing power in the hands of signers `cancelledSigs` like mechanics can be added to the `cancel()` logic.

Currently `cancelSig()` has no implications on `cancel()` which doesn't look like a consistent behavior as whenever a signer runs `cancelSig()` they could not expect that it's not enough and there is also `cancel()` that needs to be run because otherwise the signature is de facto continued to be used as fully valid.

At the same time, the ability of any signer to cancel unconditionally appears to be excessive. Why is the cancellation justified when there are enough votes to pass the proposal threshold even without this signature? We see a griefing surface here and not too much value added.

Therefore, we are not convinced that the proposed fix mitigates the described risk completely.

5.1.2 Potential Denial of Service (DoS) attack on `NounsAuctionHouseFork` Contract

Severity: High Risk

Context: NounsAuctionHouseFork.sol#L213

Description: The potential vulnerability arises during the initialization of the `NounsAuctionHouseFork` contract, which is deployed and initialized via the `executeFork()` function when a new fork is created. At this stage, the state variable startNounId within the `NounsTokenFork` contract is set corresponding to the `nounId` currently being auctioned in the `NounsAuctionHouse`.

It should be noted that the `NounsAuctionHouseFork` contract is initially in a paused state and requires a successful proposal to unpause it, thus enabling the minting of new nouns tokens within the fork.

Based on the current structure, an attacker can execute a DoS attack through the following steps:

1. Assume the `executeFork()` threshold is 7 nouns and the attacker owns 8 nouns. The current `nounId` being auctioned is 735.
2. The attacker places the highest bid for `nounId` 735 in the `NounsAuctionHouse` contract and waits for the auction's conclusion.
3. Once the auction concludes, the attacker calls `escrowToFork()` with his 8 nouns, triggering the `executeFork()` threshold.
4. Upon invoking `executeFork()`, new fork contracts are deployed. Below is the state of both `NounsAuctionHouseFork` and `NounsAuctionHouse` contracts at this juncture:

```
NounsAuctionHouseFork state:
    nounId -> 0
    amount -> 0
    startTime -> 0
    endTime -> 0
    bidder -> 0x0000000000000000000000000000000000000000
    settled -> false

NounsAuctionHouse state:
    nounId -> 735
    amount -> 50000000000000000000
    startTime -> 1686014675
    endTime -> 1686101075
    bidder -> 0xE6b3367318C5e11a6eED3Cd0D850eC06A02E9b90 (attacker's address)
    settled -> false
```

5. The attacker executes `settleCurrentAndCreateNewAuction()` on the `NounsAuctionHouse` contract, thereby acquiring the `nounId` 735.
6. Following this, the attacker invokes `joinFork()` on the main DAO and joins the fork with `nounId` 735. This action effectively mints `nounId` 735 within the fork and subsequently triggers a DoS state in the `NounsAuctionHouseFork` contract.
7. At a later time, a proposal is successfully passed and the `unpause()` function is called on the `NounsAuctionHouseFork` contract.
8. A revert occurs when the `_createAuction()` function tries to mint `tokenId` 735 in the fork (which was already minted during the `joinFork()` call), thus re-pausing the contract.

More broadly, this could happen if the buyer of the fork DAO's `startNounId` (and successive ones) on the original DAO (i.e. the first Nouns that get auctioned after a fork is executed) joins the fork with those tokens, even without any malicious intent, before the fork's auction is unpaused by its governance. Applying of delayed governance on fork DAO makes this timing-based behavior more feasible. One has to buy one or more of the original DAO tokens auctioned after the fork was executed and use them to join the fork immediately.

The `NounsAuctionHouseFork` contract gets into a DoS state, necessitating a contract update in the `NounsTokenFork` contract to manually increase the `_currentNounId` state variable to restore the normal flow in the `NounsAuctionHouseFork`.

High likelihood + Medium impact = High Severity.

Likelihood: High, because its a very likely scenario to happen, even unintentionally, the scenario can be triggered by a non-malicious user that just wants to join the fork with a fresh bought Noun from the auction house. Impact: Medium, because forking is bricked for at least several weeks until the upgrade proposal passes and is in place. This is not simply having a contract disabled for a period of time, this can be considered as a loss of assets for the Forked DAO as well, i.e. imagine that the Forked DAO needs funding immediately. On top of this, the contract upgrade would have to be done on the `NounsTokenFork` contract to correct the `_currentNounId` state variable to a valid value and fix the Denial of Service in the `NounsAuctionHouseFork`. Would the fork joiners be willing to perform such a risky update in such a critical contract?

Recommendation: Consider one of the below options:

1. Update `NounsTokenFork._currentNounId` in `claimDuringForkPeriod()` if `tokenIds[i] >= _currentNounId` so that fork auctions begin on Nouns that have not been auctioned (and migrated+minted on this fork) on the original DAO yet.
2. Store the initial `nounId` that is being auctioned when the `executeFork()` function is invoked, and then prevent any subsequent calls to the `joinFork()` function with a `nounId` that is equal to or higher than that value. This will ensure that the `_createAuction()` function does not attempt to mint an already minted tokenId, thus preventing the contract from getting into a DoS state. However, note that with `MAX_FORK_PERIOD` being 14 days which means, depending on the duration of delayed governance and status of unclaimed escrow tokens, this could prevent up to 14 auctioned Nouns in the worst case from joining the fork.

Nouns:

Mitigation PR here: PR 714.

Spearbit: Verified that PR: PR 714 fixes the issue using the first recommendation.

5.2 Medium Risk

5.2.1 Total supply can be low down to zero after the fork, allowing for execution of exploiting proposals from any next joiners

Severity: Medium Risk

Context: NounsDAOLogicV1Fork.sol#L242-L305

Description: Total supply can be low down to reaching zero during forking period, so any holder then entering forked DAO with `joinFork()` can push manipulating proposals and force all the later joiners either to rage quit or to be exploited.

As an example, if there is a group of nouns holders that performed fork for pure financial reasons, all claimed forked nouns and quitted. Right after that it is `block.timestamp < forkingPeriodEndTimestamp`, so `isForkPeriodActive(ds) == true` in original DAO contract.

In the same time forked token's `adjustedTotalSupply` is zero (all new tokens were sent to timelock):

- NounsDAOLogicV1Fork.sol#L201-L208

```
    function quit(uint256[] calldata tokenIds) external nonReentrant {
        ...
        for (uint256 i = 0; i < tokenIds.length; i++) {
>>          nouns.transferFrom(msg.sender, address(timelock), tokenIds[i]);
        }
```

- NounsDAOLogicV1Fork.sol#L742-L744

```
    function adjustedTotalSupply() public view returns (uint256) {
        return nouns.totalSupply() - nouns.balanceOf(address(timelock));
    }
```

Also, `NounsTokenFork.remainingTokensToClaim() == 0`, so `checkGovernanceActive()` check does not revert in the forked DAO contract:

- NounsDAOLogicV1Fork.sol#L346-L349)

```
    function checkGovernanceActive() internal view {
        if (block.timestamp < delayedGovernanceExpirationTimestamp && nouns.remainingTokensToClaim() >
        ↪ 0)
            revert WaitingForTokensToClaimOrExpiration();
    }
```

Original DAO holders can enter new DAO with `joinFork()` only, that will keep `checkGovernanceActive()` non-reverting in the forked DAO contract:

- NounsDAOV3Fork.sol#L139-L158

```
    function joinFork(
        NounsDAOStorageV3.StorageV3 storage ds,
        uint256[] calldata tokenIds,
        uint256[] calldata proposalIds,
        string calldata reason
    ) external {
        ...

        for (uint256 i = 0; i < tokenIds.length; i++) {
            ds.nouns.transferFrom(msg.sender, timelock, tokenIds[i]);
        }

>>      NounsTokenFork(ds.forkDAOToken).claimDuringForkPeriod(msg.sender, tokenIds);

        emit JoinFork(forkEscrow.forkId() - 1, msg.sender, tokenIds, proposalIds, reason);
    }
```

As `remainingTokensToClaim` stays zero as `claimDuringForkPeriod()` doesn't affect it:

- NounsTokenFork.sol#L166-L174

```
    function claimDuringForkPeriod(address to, uint256[] calldata tokenIds) external {
        if (msg.sender != escrow.dao()) revert OnlyOriginalDAO();
        if (block.timestamp > forkingPeriodEndTimestamp) revert OnlyDuringForkingPeriod();

        for (uint256 i = 0; i < tokenIds.length; i++) {
            uint256 nounId = tokenIds[i];
            _mintWithOriginalSeed(to, nounId);
        }
    }
```

In this situation both quorum and proposal thresholds will be zero, proposals can be created with `creationBlock = block.number`, at which only recently joined holder have voting power:

- NounsDAOLogicV1Fork.sol#L242-L305

```
    function propose(
        address[] memory targets,
        uint256[] memory values,
        string[] memory signatures,
        bytes[] memory calldatas,
        string memory description
    ) public returns (uint256) {
        checkGovernanceActive();

        ProposalTemp memory temp;

        temp.totalSupply = adjustedTotalSupply();

>>      temp.proposalThreshold = bps2Uint(proposalThresholdBPS, temp.totalSupply);

        require(
            nouns.getPriorVotes(msg.sender, block.number - 1) > temp.proposalThreshold,
            'NounsDAO::propose: proposer votes below proposal threshold'
        );
        ...
>>      newProposal.proposalThreshold = temp.proposalThreshold;
>>      newProposal.quorumVotes = bps2Uint(quorumVotesBPS, temp.totalSupply);
        ...
>>      newProposal.creationBlock = block.number;
```

- DeployDAOV3NewContractsBase.s.sol#L18-L23

```
contract DeployDAOV3NewContractsBase is Script {
    ...
    uint256 public constant FORK_DAO_PROPOSAL_THRESHOLD_BPS = 25; // 0.25%
    uint256 public constant FORK_DAO_QUORUM_VOTES_BPS = 1000; // 10%
```

This will give the first joiner the full power over all the later joiners:

- NounsDAOLogicV1Fork.sol#L577-L589

```
    function castVoteInternal(
        address voter,
        uint256 proposalId,
        uint8 support
    ) internal returns (uint96) {
        ...

        /// @notice: Unlike GovernerBravo, votes are considered from the block the proposal was created
        ↪  in order to normalize quorumVotes and proposalThreshold metrics
>>      uint96 votes = nouns.getPriorVotes(voter, proposal.creationBlock);
```

Say if Bob, the original nouns DAO holder with `1` noun, joined when total supply was zero, he can create proposals and with regard for these proposals his only vote will be `100`% of the DAO voting power.

Bob can create a proposal to transfer all the funds to himself or a hidden malicious one like shown in *Fork escrowers can exploit the fork or force late joiners to quit* step 6. All the later joiners will not be able to stop this proposal, no matter how big their voting power be, as votes will be counted as on block where Bob had `100`% of the votes.

As the scenario above is a part of expected workflow (i.e. all fork initiators can be reasonably expected to quit fast enough), the probability of it is medium, while the probability of inattentive late joiners being exploited by Bob's proposal is medium too (there is not much time to react and some holders might first of all want to explore new fork functionality), so overall probability is low, while the impact is full loss of funds for such joiners.

Per low combined likelihood and high impact setting the severity to be medium.

Recommendation: In order to achieve stability forked DAO needs to secure enough members first, so, as one of the options, fork period might be required to be concluded prior to any proposals can be made in the new DAO.

For this end consider additionally passing the `forkingPeriodEndTimestamp` to forked DAO (now it is only passed to the forked token), and forbidding new proposals until `forkingPeriodEndTimestamp`, for example:

- NounsDAOStorageV1Fork.sol#L51-L54)

```
    /// @notice The latest proposal for each proposer
    mapping(address => uint256) public latestProposalIds;

    uint256 public delayedGovernanceExpirationTimestamp;

+   /// @notice The forking period expiration timestamp, after which new tokens cannot be claimed by
↪   the original DAO
+   uint256 public forkingPeriodEndTimestamp;
```

- NounsDAOLogicV1Fork.sol#L164-L193

```
    function initialize(
        address timelock_,
        address nouns_,
        uint256 votingPeriod_,
        uint256 votingDelay_,
        uint256 proposalThresholdBPS_,
        uint256 quorumVotesBPS_,
        address[] memory erc20TokensToIncludeInQuit_,
        uint256 delayedGovernanceExpirationTimestamp_,
+       uint256 forkingPeriodEndTimestamp_
    ) public virtual {
        __ReentrancyGuard_init_unchained();
        ...
        delayedGovernanceExpirationTimestamp = delayedGovernanceExpirationTimestamp_;
+       forkingPeriodEndTimestamp = forkingPeriodEndTimestamp_;
    }
```

- NounsDAOLogicV1Fork.sol#L107

```
    error WaitingForTokensToClaimOrExpiration();
+   error WaitingForForkPeriodEnd();
```

- NounsDAOLogicV1Fork.sol#L242-L249)

```
    function propose(
        address[] memory targets,
        uint256[] memory values,
        string[] memory signatures,
        bytes[] memory calldatas,
        string memory description
    ) public returns (uint256) {
        checkGovernanceActive();
+       if (block.timestamp <= forkingPeriodEndTimestamp) {
+           revert WaitingForForkPeriodEnd();
+       }
```

Nouns:

Fix implementation: nounsDAO/nouns-monorepo#717.

Spearbit: The fix looks good.

5.2.2 Duplicate ERC20 tokens will send a greater than prorata token share leading to loss of DAO funds

Severity: Medium Risk

Context: NounsDAOV3Admin.sol#L497-L507 NounsDAOV3Fork.sol#L224-L230 NounsDAOLogicV1Fork.sol#L717-L721 NounsDAOLogicV1Fork.sol#L210-L215

Description: `_setErc20TokensToIncludeInFork()` is an admin function for setting ERC20 tokens that are used when splitting funds to a fork. However, there are no sanity checks for duplicate ERC20 tokens in the `erc20tokens` parameter. While `STETH` is the only ERC20 token applicable for now, it is conceivable that DAO treasury may include others in future.

The same argument applies to `_setErc20TokensToIncludeInQuit()` and members quitting from the fork DAO.

Duplicate tokens in the array will send a greater than prorata share of those tokens to the fork DAO treasury in `sendProRataTreasury()` or to the quitting member in `quit()`. This will lead to loss of funds for the original DAO and fork DAO respectively.

Low likelihood + High impact = Medium severity.

Recommendation: Consider adding a check to filter out any duplicates in the setters or in `sendProRataTreasury()/quit()`.

Nouns:

Fix PR: PR 733.

Spearbit: Verified that PR 733 fixes this issue as recommended using checks in setters.

5.2.3 A malicious proposer can create arbitrary number of maliciously updatable proposals to significantly grief the protocol

Severity: Medium Risk

Context: NounsDAOV3Proposals.sol#L783-L798 NounsDAOV3Proposals.sol#L171 NounsDAOV3Proposals.sol#L818-L823 NounsDAOV3Proposals.sol#L269-L423

Description: `checkNoActiveProp()` is documented as: "*This is a spam protection mechanism to limit the number of proposals each noun can back.*" However, this mitigation applies to proposer addresses holding Nouns but not the Nouns themselves because `checkNoActiveProp()` relies on checking the state of proposals tracked by proposer via `latestProposalId = ds.latestProposalIds[proposer]`. A malicious proposer can move (transfer/delegate) their Noun(s) to different addresses for circumventing this mitigation and create proposals from those new addresses to spam. Furthermore, proposal updation in the protocol does not check for the proposer meeting any voting power threshold at the time of updation.

A malicious proposer can create arbitrary number of proposals, each from a different address by transferring/delegating their Nouns, and then update any/all of them to be malicious. Substantial effort will be required to differentiate all such proposals from the authentic ones and then cancel them, leading to DAO governance DoS griefing.

Medium likelihood + Medium impact = Medium severity.

Recommendation: Consider:

1. A redesign where proposal creation spam mitigation is not based on the Noun controlling address but the Noun itself.
2. Adding proposal threshold check for voting power during updation.

Nouns:

> Won't Fix. This is a known issue from the launch of Nouns, and is mitigated by canceling proposals once proposer doesn't have enough balance to meet threshold, as well as the vetoer in extreme cases. Such spammy behavior should easily become suspicious and token holders will be ready to cancel all such proposals, same as they are today.
>
> Specifically, when it comes to obvious spamming, we don't think the updatable proposals adds any meaningful risk. The spammy behavior is the main red flag and using this new feature with more stealth is already discussed in "*A malicious proposer can update proposal past inattentive voters to sneak in otherwise unacceptable details*".

Spearbit: Acknowledged.

5.2.4 A malicious proposer can update proposal past inattentive voters to sneak in otherwise unacceptable details

Severity: Medium Risk

Context: NounsDAOV3Proposals.sol#L269-L423 NounsDAOV3Admin.sol#L118 NounsDAOV3Votes.sol#L70-L293

Description: Updatable proposal description and transactions is a new feature being introduced in V3 to improve the UX of the proposal flow to allow proposal editing on-chain. The motivation for this feature as described in the spec is: "*Proposals get voter feedback almost entirely only once they are on-chain. At the same time, proposers are relunctant to cancel and resubmit their proposals for multiple reasons, e.g. prefering to avoid restarting the proposal lifecycle and thus delay funding*."

However, votes are bound only to the proposal identifier and not to their description (which describes the motivation/intention/usage etc.) or the transactions (values transferred, contracts/functions of interaction etc.).

Inattentive voters may (decide to) cast their votes based on a stale proposal's description/transactions which could since have been updated. For example, someone voting Yes on the initial proposal version may vote No if they see the updated details. A very small voting delay (`MIN_VOTING_DELAY` is 1 block) may even allow a malicious proposer to sneak in a malicious update at the very end of the updatable period so that voters do not see it on time to change their votes being cast. Delays in front-ends updating the proposal details may contribute to this scenario.

A malicious proposer updates proposal with otherwise unacceptable txs/description to get support of inattentive voters who cast their votes based on acceptable older proposal versions. Malicious proposal passes to transfer a significant amount of treasury to unauthorized receivers for unacceptable reasons.

Low likelihood + High impact = Medium severity.

Recommendation: Increase minimum threshold for voting delay to a reasonable value much greater than the current 1 block (12 seconds) e.g. few days which always (i.e. even assuming that DAO can change voting delay to any allowed value) gives sufficient time for voters to notice/evaluate updated proposal details after updatable period ends and voting period is active. A more defensive design is to make the votes binding on proposal description+transactions instead of only the proposal identifier.

Nouns:

> Won't Fix.
>
> We think it highly unlikely the DAO will set voting delay to be very short. The DAO enjoys a multi-day period it uses to make sense of proposals.

Spearbit: Acknowledged.

5.2.5 NounsDAOLogicV1Fork's `quit()` performing external calls in-between total supply and balance reads can allow for treasury funds stealing via cross-contract reentrancy

Severity: Medium Risk

Context: NounsDAOLogicV1Fork.sol#L201-L222

Description: Let's suppose there is an initiative group of nouns holders that performed fork, claimed and immediately quitted (say for pure financial reasons). Right after that it is `block.timestamp < forkingPeriodEndTimestamp`, so `isForkPeriodActive(ds) == true` in original DAO contract, while `NounsTokenFork.remainingTokensToClaim() == 0`, so `checkGovernanceActive()` doesn't revert in the forked DAO contract, which have no material holdings.

For simplicity let's say there is Bob and Alice, both aren't part of this group and still are in the original DAO, Bob have 2 nouns, Alice have 1, each nouns' share of treasury is `1 stETH` and `100 ETH`, `erc20TokensToIncludeInQuit = [stETH]`.

All the above are going concern assumptions (a part of expected workflow), let's now have a low probability one: stETH contract was upgraded and now performs `_beforetokentransfer()` callback on every transfer to a destination address as long as it's a contract (i.e. it has a callback, for simplicity let's assume it behaves similarly

to ERC-721 `safeTransfer`). It doesn't make it malicious or breaks IERC20, let's just suppose there is a strong enough technical reason for such an upgrade.

If Alice now decides to join this fork, Bob can steal from her:

1. Alice calls NounsDAOV3's `joinFork()`, `1 stETH` and `100 ETH` is transferred to NounsDAOLogicV1Fork:

- NounsDAOV3Fork.sol#L139-L158

```
function joinFork(
    NounsDAOStorageV3.StorageV3 storage ds,
    uint256[] calldata tokenIds,
    uint256[] calldata proposalIds,
    string calldata reason
) external {
    if (!isForkPeriodActive(ds)) revert ForkPeriodNotActive();

    INounsDAOForkEscrow forkEscrow = ds.forkEscrow;
    address timelock = address(ds.timelock);
    sendProRataTreasury(ds, ds.forkDAOTreasury, tokenIds.length, adjustedTotalSupply(ds));

    for (uint256 i = 0; i < tokenIds.length; i++) {
        ds.nouns.transferFrom(msg.sender, timelock, tokenIds[i]);
    }

    NounsTokenFork(ds.forkDAOToken).claimDuringForkPeriod(msg.sender, tokenIds);

    emit JoinFork(forkEscrow.forkId() - 1, msg.sender, tokenIds, proposalIds, reason);
}
```

Alice is minted 1 forked noun:

- NounsTokenFork.sol#L166-L174)

```
function claimDuringForkPeriod(address to, uint256[] calldata tokenIds) external {
    if (msg.sender != escrow.dao()) revert OnlyOriginalDAO();
    if (block.timestamp > forkingPeriodEndTimestamp) revert OnlyDuringForkingPeriod();

    for (uint256 i = 0; i < tokenIds.length; i++) {
        uint256 nounId = tokenIds[i];
        _mintWithOriginalSeed(to, nounId);
    }
}
```

2. Bob transfers all to attack contract (cBob), that joins the DAO with 1 noun. Forked treasury is `2 stETH` and `200 ETH`, cBob and Alice both have 1 noun.
3. cBob calls quit() and reenters NounsDAOV3's `joinFork()` on stETH `_beforetokentransfer()` (and nothing else):

- NounsDAOLogicV1Fork.sol#L201-L222

```
    function quit(uint256[] calldata tokenIds) external nonReentrant {
        checkGovernanceActive();

        uint256 totalSupply = adjustedTotalSupply();

        for (uint256 i = 0; i < tokenIds.length; i++) {
            nouns.transferFrom(msg.sender, address(timelock), tokenIds[i]);
        }

        for (uint256 i = 0; i < erc20TokensToIncludeInQuit.length; i++) {
            IERC20 erc20token = IERC20(erc20TokensToIncludeInQuit[i]);
            uint256 tokensToSend = (erc20token.balanceOf(address(timelock)) * tokenIds.length) /
            ↪  totalSupply;
>>          bool erc20Sent = timelock.sendERC20(msg.sender, address(erc20token), tokensToSend);
            if (!erc20Sent) revert QuitERC20TransferFailed();
        }

        uint256 ethToSend = (address(timelock).balance * tokenIds.length) / totalSupply;
        bool ethSent = timelock.sendETH(msg.sender, ethToSend);
        if (!ethSent) revert QuitETHTransferFailed();

        emit Quit(msg.sender, tokenIds);
    }
```

4. cBob have joined fork with another noun, stETH transfer concludes. Forked treasury is `2 stETH` and `300 ETH`, while `1 stETH` was just sent to cBob.
5. With `quit()` resumed, `(address(timelock).balance * tokenIds.length) / totalSupply = (300 * 1) / 2 = 150 ETH` is sent to cBob:

- NounsDAOLogicV1Fork.sol#L201-L222

```
    function quit(uint256[] calldata tokenIds) external nonReentrant {
        checkGovernanceActive();

        uint256 totalSupply = adjustedTotalSupply();

        for (uint256 i = 0; i < tokenIds.length; i++) {
            nouns.transferFrom(msg.sender, address(timelock), tokenIds[i]);
        }

        for (uint256 i = 0; i < erc20TokensToIncludeInQuit.length; i++) {
            IERC20 erc20token = IERC20(erc20TokensToIncludeInQuit[i]);
            uint256 tokensToSend = (erc20token.balanceOf(address(timelock)) * tokenIds.length) /
            ↪  totalSupply;
            bool erc20Sent = timelock.sendERC20(msg.sender, address(erc20token), tokensToSend);
            if (!erc20Sent) revert QuitERC20TransferFailed();
        }

>>      uint256 ethToSend = (address(timelock).balance * tokenIds.length) / totalSupply;
        bool ethSent = timelock.sendETH(msg.sender, ethToSend);
        if (!ethSent) revert QuitETHTransferFailed();

        emit Quit(msg.sender, tokenIds);
    }
```

6. Forked treasury is `2 stETH` and `150 ETH`, cBob calls `quit()` again without reentering (say on zero original nouns balance condition), obtains `1 stETH` and `75 ETH`, the same is left for Alice.

Bob stole `25 ETH` from Alice.

Attacking function logic can be as simple as {`quit()` as long as there is forkedNoun on my balance, perform `joinFork()` on the callback as long as there is noun on my balance}.

Alice lost a part of treasury funds. The scale of the steps above can be increased to drain more significant value in absolute terms.

Per low likelihood and high principal funds loss impact setting the severity to be medium.

Recommendation: Core issue is that state variables, nouns total supply and treasury asset balances, are obtained before and after the external calls. This can be fixed by gathering the state beforehand, for example:

- NounsDAOLogicV1Fork.sol#L201-L222

```
    function quit(uint256[] calldata tokenIds) external nonReentrant {
        checkGovernanceActive();

        uint256 totalSupply = adjustedTotalSupply();
+       uint256 erc20length = erc20TokensToIncludeInQuit.length;
+       uint256[] memory balances = new uint256[](erc20length + 1);
+       IERC20[] memory erc20tokens = new IERC20[](erc20length);
+       for (uint256 i = 0; i < erc20length; i++) {
+           erc20tokens[i] = IERC20(erc20TokensToIncludeInQuit[i]);
+           balances[i] = erc20tokens[i].balanceOf(address(timelock));
+       }
+       balances[erc20length] = address(timelock).balance;

        for (uint256 i = 0; i < tokenIds.length; i++) {
            nouns.transferFrom(msg.sender, address(timelock), tokenIds[i]);
        }

-       for (uint256 i = 0; i < erc20TokensToIncludeInQuit.length; i++) {
+       for (uint256 i = 0; i < erc20length; i++) {
-           IERC20 erc20token = IERC20(erc20TokensToIncludeInQuit[i]);
-           uint256 tokensToSend = (erc20token.balanceOf(address(timelock)) * tokenIds.length) /
↪   totalSupply;
-           bool erc20Sent = timelock.sendERC20(msg.sender, address(erc20token), tokensToSend);
+           uint256 tokensToSend = (balances[i] * tokenIds.length) / totalSupply;
+           bool erc20Sent = timelock.sendERC20(msg.sender, address(erc20tokens[i]), tokensToSend);
            if (!erc20Sent) revert QuitERC20TransferFailed();
        }

-       uint256 ethToSend = (address(timelock).balance * tokenIds.length) / totalSupply;
+       uint256 ethToSend = (balances[erc20length] * tokenIds.length) / totalSupply;
        bool ethSent = timelock.sendETH(msg.sender, ethToSend);
        if (!ethSent) revert QuitETHTransferFailed();

        emit Quit(msg.sender, tokenIds);
    }
```

Nouns: Fixed in PR 722.

Spearbit: Fix looks good.

5.2.6 A malicious DAO can mint arbitrary fork DAO tokens

Severity: Medium Risk

Context: NounsDAOV3Proposals.sol#L495 NounsDAOV3Fork.sol#L203-L205 NounsTokenFork.sol#L166-L174

Description: The original DAO is assumed to be honest during the fork period which is reinforced in the protocol by preventing it from executing any malicious proposals during that time. Fork joiners are minted fork DAO tokens by the original DAO via `claimDuringForkPeriod()` which enforces the fork period on the fork DAO side. However, the notion of fork period is different on the fork DAO compared to the original DAO (as described in Issue 16), i.e. while original DAO excludes `forkEndTimestamp` from the fork period, fork DAO includes `forkingPeriodEndTimestamp` in its notion of the fork period.

If the original DAO executes a malicious proposal exactly in the block at `forkEndTimestamp` which makes a call to `claimDuringForkPeriod()` to mint arbitrary fork DAO tokens then the proposal will succeed on the original DAO side because it is one block beyond its notion of fork period. The `claimDuringForkPeriod()` will succeed on the fork DAO side because it is in the last block in its notion of fork period. The original DAO therefore can successfully mint arbitrary fork DAO tokens which can be used to: 1) brick the fork DAO when those tokens are attempted to be minted via auctions later or 2) manipulate the fork DAO governance to steal its treasury funds.

In PoS, blocks are exactly 12 seconds apart. With `forkEndTimestamp = block.timestamp + ds.forkPeriod;` and `ds.forkPeriod` now set to `7 days`, `forkEndTimestamp` is exactly 50400 blocks (`7*24*60*60/12`) after the block in which `executeFork()` was executed. A malicious DAO can coordinate to execute such a proposal exactly in that block.

Low likelihood + High impact = Medium severity.

Recommendation: Make the treatment of fork period consistent between the original and fork DAOs in:

- NounsTokenFork.sol#L166-L174)

```
    function claimDuringForkPeriod(address to, uint256[] calldata tokenIds) external {
        if (msg.sender != escrow.dao()) revert OnlyOriginalDAO();
-       if (block.timestamp > forkingPeriodEndTimestamp) revert OnlyDuringForkingPeriod();
+       if (block.timestamp >= forkingPeriodEndTimestamp) revert OnlyDuringForkingPeriod();

        for (uint256 i = 0; i < tokenIds.length; i++) {
            uint256 nounId = tokenIds[i];
            _mintWithOriginalSeed(to, nounId);
        }
    }
```

Nouns:

Implemented fix: nounsDAO/nouns-monorepo#719.

Spearbit: Verified that PR 719 fixes the issue as recommended.

5.2.7 Inattentive fork escrowers may lose funds to fork quitters

Severity: Medium Risk

Context: Fork-Spec NounsTokenFork.sol#L148-L157 NounsDAOLogicV1Fork.sol#L346-L349 NounsDAOLogicV1Fork.sol#L201-L222

Description: Fork escrowers already have their original DAO treasury pro rate funds transferred to the fork DAO treasury (when fork executes) and are expected to `claimFromEscrow()` after fork executes to mint their fork DAO tokens and thereby lay claim on their pro rata share of fork DAO treasury for governance or exiting. Inattentive fork escrowers who fail to do so will force a delayed governance of 30 days (currently proposed value) on the fork DAO and beyond that will allow fork DAO members to quit with a greater share of the fork DAO treasury because fork execution transfers all escrowers' original DAO treasury funds to fork DAO treasury.

Inattentive slow-/non-claiming fork escrowers may lose funds to quitters if they do not claim their fork DAO tokens before its governance is active in 30 days after fork executes. They will also be unaccounted for in DAO functions like quorum and proposal threshold.

While we would expect fork escrowers to be attentive and claim their fork DAO tokens well within the delayed governance period, the protocol design can be more defensive of slow-/non-claimers by protecting their funds on the fork DAO from quitters.

Low likelihood + High impact = Medium severity.

Recommendation: Consider changing `NounsDAOLogicV1Fork.adjustedTotalSupply()` to be `nouns.totalSupply() - nouns.balanceOf(address(timelock)) + nouns.remainingTokensToClaim()` which will include any unclaimed tokens in fork DAO calculations. This will effectively freeze the slow-/non-claiming fork escrowers' funds within the fork DAO treasury instead of them losing it to quitters (outside of any executed proposals acting on treasury funds) if they do not claim within 30 days.

Non-claimers will permanently affect the fork DAO supply which affects proposals/quitting but that is valid because unclaimed original DAO escrowed tokens have a perpetual 1:1 right on equivalent fork DAO tokens which should be accounted for correctly.

Nouns:

> Acknowledged, and here's a fix PR: nounsDAO/nouns-monorepo#723.

Spearbit: Verified that PR 723 fixes the issue as recommended.

5.2.8 Upgrading timelock without transferring the nouns from old timelock balance will increase adjusted total supply

Severity: Medium Risk

Context: ProposeTimelockMigrationCleanupMainnet.s.sol#L94

Description: There is one noun on timelock V1 balance, and can be others as of migration time:

- etherscan.io/token/0x9c8ff314c9bc7f6e59a9d9225fb22946427edc03?a=0x0BC3807Ec262cB779b38D65b38158acC3bfe

Changing `ds.timelock` without nouns transfer will increase adjusted total supply:

- NounsDAOV3Fork.sol#L199-L201

```
function adjustedTotalSupply(NounsDAOStorageV3.StorageV3 storage ds) internal view returns
↪  (uint256) {
    return ds.nouns.totalSupply() - ds.nouns.balanceOf(address(ds.timelock)) -
    ↪  ds.forkEscrow.numTokensOwnedByDAO();
}
```

As of time of this writing adjustedTotalSupply() will be increased by `1` due to treasury token reclassification, the upgrade will cause a `(13470 + 14968) * 1733.0 * (1 / 742 - 1 / 743) = 89 USD` loss per noun or `(13470 + 14968) * 1733.0 / 743 = 66330 USD` cumulatively for all nouns holders.

Per high likelihood and low impact setting the severity to be medium.

Recommendation: Consider adding to `ProposeTimelockMigrationCleanupMainnet` the step of moving the treasury nouns from timelock V1 to timelock V2 contract to keep `adjustedTotalSupply()` reading unchanged.

Nouns: Fixed in PR 721.

Spearbit: Fix looks okay. Conditional on that `timelockV1` owns no other nouns besides `687` as of time of the upgrade.

5.2.9 Fork escrowers can exploit the fork or force late joiners to quit

Severity: Medium Risk

Context: NounsDAOLogicV1Fork.sol#L347

Description: Based in the current supply of Nouns and the following parameters that will be used during the upgrade to V3:

- `Nouns total supply`: 743
- `forkThresholdBPS_`: 2000 (20%)
- `forkThreshold`: 148, hence 149 Nouns need to be escrowed to be able to call `executeFork()`

The following attack vector would be possible:

1. Attacker escrows 75 tokens.
2. Bob escrows 74 tokens to reach the `forkThreshold`.
3. Bob calls `executeFork()` and `claimFromEscrow()`.
4. Attacker calls `claimFromEscrow()` right away. As now `nouns.remainingTokensToClaim()` is zero the governance is now active and proposals can be created.
5. Attacker creates a malicious proposal. Currently the attacker has 75 Nouns and Bob 74 in the fork. This means that the attacker has the majority of the voting power and whatever he proposes can not be denied.
   - `NounsForkToken.getPriorVotes(attacker, <proposalCreationBlock>) -> 75`
   - `NounsForkToken.getPriorVotes(Bob , <proposalCreationBlock>) -> 74`
6. The proposal is created with the following description: *"Proposal created to upgrade the `NounsAuctionHouseFork` to a new implementation similar to the main `NounsAuctionHouse`"*. The attacker deploys this new implementation and simply performs the following change in the code:

```
    modifier initializer() {
-       require(_initializing || !_initialized, "Initializable: contract is already initialized");
+       require(!_initializing || _initialized, "Initializable: contract is already initialized");

        bool isTopLevelCall = !_initializing;
        if (isTopLevelCall) {
            _initializing = true;
            _initialized = true;
        }

        _;

        if (isTopLevelCall) {
            _initializing = false;
        }
    }
```

The proposal is created with the following data:

```
targets[0] = address(contract_NounsAuctionHouseFork);
values[0] = 0;
signatures[0] = 'upgradeTo(address)';
calldatas[0] = abi.encode(address(contract_NounsAuctionHouseForkExploitableV1));
```

7. Proposal is created and is now in Pending state. During the next days, users keep joining the fork increasing the funds of the fork treasury as the fork period is still active.
8. 5 days later proposal is in Active state and the attacker votes to pass it. Bob, who does not like the proposal, votes to reject it.

- `quorumVotes`: 14
- `forVotes`: 75
- `againstVotes`: 74

9. As the attacker and Bob were the only users that had any voting power at the time of proposal creation, five days later, the proposal is successful.
10. Proposal is queued.
11. 3 weeks later proposal is executed.
12. The `NounsAuctionHouseFork` contract is upgraded to the malicious version and the attacker re-initialize it and sets himself as the owner:

```
contract_NounsAuctionHouseFork.initialize(attacker, NounsForkToken, <WETH address>, 0, 0, 0, 0)
```

13. Attacker, who is now the owner, upgrades the `NounsAuctionHouseFork` contract, once again, to a new implementation that implements the following function:

```
    function burn(uint256[] memory _nounIDs) external onlyOwner{
        for (uint256 i; i < _nounIDs.length; ++i){
            nouns.burn(_nounIDs[i]);
        }
    }
```

14. Attacker now, burns all the Nouns Tokens in the fork except the ones that he owns.
15. Attacker calls `quit()` draining the whole treasury:

```
  NounsTokenFork.totalSupply() -> 75
  attacker.balance -> 0
  contract_stETH.balanceOf(attacker) -> 0
  forkTreasury.balance -> 2005_383580080753701211
  contract_stETH.balanceOf(forkTreasury) -> 2005_383580080753701210

attacker calls -> contract_NounsDAOLogicV1Fork.quit([0, ... 74])

  attacker.balance -> 2005_383580080753701211
  contract_stETH.balanceOf(attacker) -> 2005_383580080753701208
  forkTreasury.balance -> 0
  contract_stETH.balanceOf(forkTreasury) -> 1
```

Basically, the condition that should be met for this exploit is that at the time of proposal creation the attacker has more than 51% of the voting power. This is more likely to happen in small forks. If this happens, users will be forced to leave or be exploited. As there is no vetoer role, noone will be able to stop this type of proposals.

Recommendation: Consider removing the `checkGovernanceActive()` function and, instead, do not allow users to create any proposal in the fork until the fork period is not active anymore or which is the same: `isForkPeriodActive == false`.

Consider also auditing any new proposal that is added to the DAO to mitigate these risks.

Nouns: Followed Spearbit's recommendation and fixed the issue in the PR 717. Proposals can not be created and users can not quit the fork until the forking period is over.

Spearbit: Acknowledged.

5.2.10 Including non-standard ERC20 tokens will revert and prevent forking/quitting

Severity: Medium Risk

Context: NounsDAOExecutorV2.sol#L223, NounsDAOV3Fork.sol#L228-L229, NounsDAOLogicV1Fork.sol#L213-L214

Description: If `erc20TokensToIncludeInFork` or `erc20TokensToIncludeInQuit` accidentally/maliciously include non-confirming ERC20 tokens, such as USDT, which do not return a boolean value on transfers then `sendProRataTreasury()` and `quit()` will revert because it expects `timelock.sendERC20()` to return `true` from the underlying ERC20 transfer call. The use of `transfer()` instead of `safeTransfer()` allows this scenario.

Low likelihood + High impact = Medium severity.

Inclusion of USDT-like tokens in protocol will revert `sendProRataTreasury()` and `quit()` to prevent forking/quitting.

Recommendation: Consider the use of `safeTransfer()` instead of `transfer()` in `sendERC20()` and remove the check in callers on the expected boolean return value.

Nouns:

> Acknowledged. Fixed in this PR 720.

Spearbit: Verified that PR 720 fixes the issue as recommended.

5.2.11 Changing `voteSnapshotBlockSwitchProposalId` after it was set allows for votes double counting

Severity: Medium Risk

Context: NounsDAOV3Admin.sol#L472-L482

Description: Now `ds.voteSnapshotBlockSwitchProposalId` can be changed after it was once set to the next proposal id, there are no restrictions on repetitive setting.

In the same time, proposal votes are counted without saving the additional information needed to reconstruct the timing and `voteSnapshotBlockSwitchProposalId` moved forward as a result of such second `_setVoteSnapshotBlockSwitchProposalId()` call will produce a situation when all the older, already cast, votes for the proposals with `old_voteSnapshotBlockSwitchProposalId <= id < new_voteSnapshotBlockSwitchProposalId` will be counted as of `proposal.startBlock`, while all the never, still to be casted, votes for the very same proposals will be counted as of `proposal.creationBlock`.

Since the voting power of users can vary in-between these timestamps, this will violate the equality of voting conditions for all such proposals. Double counting will be possible and total votes greater than total supply can be cast this way: say Bob has transferred his nouns to Alice between `proposal.startBlock` and `proposal.creationBlock`, Alice voted before the change, Bob voted after the change. Bob's nounces will be counted twice.

Severity is medium: impact looks to be high, a violation of equal foot voting paves a way for voting manipulations, but there is a low likelihood prerequisite of passing a proposal for the second update for the `voteSnapshotBlockSwitchProposalId`. The latter can happen as a part of bigger pack of changes. `_setVoteSnapshotBlockSwitchProposalId()` call do not have arguments and by itself repeating it doesn't look incorrect.

Recommendation: Consider controlling the one time switch nature of this parameter directly in the code, for example:

- NounsDAOV3Admin.sol#L34-L35

```
    error InvalidProposalUpdatablePeriodInBlocks();
+   error VoteSnapshotSwitchAlreadySet();
```

- NounsDAOV3Admin.sol#L472-L482

```
    function _setVoteSnapshotBlockSwitchProposalId(NounsDAOStorageV3.StorageV3 storage ds) external
    ↪  onlyAdmin(ds) {
-       uint256 newVoteSnapshotBlockSwitchProposalId = ds.proposalCount + 1;
        uint256 oldVoteSnapshotBlockSwitchProposalId = ds.voteSnapshotBlockSwitchProposalId;
+       if (oldVoteSnapshotBlockSwitchProposalId > 0) {
+           revert VoteSnapshotSwitchAlreadySet();
+       }

+       uint256 newVoteSnapshotBlockSwitchProposalId = ds.proposalCount + 1;

        ds.voteSnapshotBlockSwitchProposalId = newVoteSnapshotBlockSwitchProposalId;

        emit VoteSnapshotBlockSwitchProposalIdSet(
            oldVoteSnapshotBlockSwitchProposalId,
            newVoteSnapshotBlockSwitchProposalId
        );
    }
```

Nouns:

Implemented recommendation: PR 718.

Spearbit: Fix looks good.

5.2.12 Key fork parameters are set outside of proposal flow, while aren't being controlled in the code

Severity: Medium Risk

Context: ForkDAODeployer.sol#L31-L81

Description: These configuration parameters are crucial for fork workflow and new DAO logic, but aren't checked when being set in `ForkDAODeployer's` constructor:

- ForkDAODeployer.sol#L31-L81

```
contract ForkDAODeployer is IForkDAODeployer {
    ...

    constructor(
        address tokenImpl_,
        address auctionImpl_,
        address governorImpl_,
        address treasuryImpl_,
        uint256 delayedGovernanceMaxDuration_,
        uint256 initialVotingPeriod_,
        uint256 initialVotingDelay_,
        uint256 initialProposalThresholdBPS_,
        uint256 initialQuorumVotesBPS_
    ) {
        ...
        delayedGovernanceMaxDuration = delayedGovernanceMaxDuration_;
        initialVotingPeriod = initialVotingPeriod_;
        initialVotingDelay = initialVotingDelay_;
        initialProposalThresholdBPS = initialProposalThresholdBPS_;
        initialQuorumVotesBPS = initialQuorumVotesBPS_;
    }
```

While most parameters are set via proposals directly and are controlled in the corresponding setters, these 5 variables are defined only once on `ForkDAODeployer` construction and neither per se visible in proposals, as `ForkDAODeployer` is being set as an address there, nor being controlled within the corresponding setters this way. Their values aren't controlled on construction either.

- NounsDAOLogicV3.sol#L820-L840

```
/**
 * @notice Admin function for setting the fork related parameters
 * @param forkEscrow_ the fork escrow contract
 * @param forkDAODeployer_ the fork dao deployer contract
 * @param erc20TokensToIncludeInFork_ the ERC20 tokens used when splitting funds to a fork
 * @param forkPeriod_ the period during which it's possible to join a fork after execution
 * @param forkThresholdBPS_ the threshold required of escrowed nouns in order to execute a fork
 */
function _setForkParams(
    address forkEscrow_,
    address forkDAODeployer_,
    address[] calldata erc20TokensToIncludeInFork_,
    uint256 forkPeriod_,
    uint256 forkThresholdBPS_
) external {
    ds._setForkEscrow(forkEscrow_);
    ds._setForkDAODeployer(forkDAODeployer_);
    ds._setErc20TokensToIncludeInFork(erc20TokensToIncludeInFork_);
    ds._setForkPeriod(forkPeriod_);
    ds._setForkThresholdBPS(forkThresholdBPS_);
}
```

- NounsDAOV3Admin.sol#L484-L495

```
/**
 * @notice Admin function for setting the fork DAO deployer contract
 */
function _setForkDAODeployer(NounsDAOStorageV3.StorageV3 storage ds, address newForkDAODeployer)
    external
    onlyAdmin(ds)
{
    address oldForkDAODeployer = address(ds.forkDAODeployer);
    ds.forkDAODeployer = IForkDAODeployer(newForkDAODeployer);

    emit ForkDAODeployerSet(oldForkDAODeployer, newForkDAODeployer);
}
```

Impact:

As an example, setting `delayedGovernanceMaxDuration = 0` bypasses NounsDAOLogicV1Fork's `checkGovernanceActive()` control and allows for stealing the whole treasury of a new forked DAO with `NounsTokenFork.claimFromEscrow() -> NounsDAOLogicV1Fork.quit()` call back-running `executeFork()` deployment transaction. An attacker will be entitled to `1 / 1 = 100%` of the new DAO funds being the only one who claimed.

Setting medium severity per low likelihood and high impact of misconfiguration, which can happen both as an operational mistake or be driven by a malicious intent.

Recommendation: Consider controlling `delayedGovernanceMaxDuration`, `initialVotingPeriod`, `initialVotingDelay`, `initialProposalThresholdBPS`, `initialQuorumVotesBPS` in the constructor to be within the hard-coded boundary values. Slight increase of size and gas cost is well justified in this setting.

Nouns:

> Acknowledged, won't fix. We disagree with the risk assessment, and like the freedom the current approach affords us.
>
> Why we think it's low risk: These parameters are actually easy to review in any proposal that sets a fork deployer, since the implementation is available onchain with these parameters set.

Spearbit: Acknowledged.

5.2.13 A malicious DAO can hold token holders captive by setting `forkPeriod` to an unreasonably low value

Severity: Medium Risk

Context: NounsDAOV3Admin.sol#L516-L524

Description: A malicious majority can reduce the number of Noun holders joining an executed fork by setting the `forkPeriod` to an unreasonably low value, e.g. 0, because there is no `MIN_FORK_PERIOD` enforced (MAX is 14 days). This in combination with an unreasonably high `forkThresholdBPS` (no min/max enforced) will allow a malicious majority to hold captive those minority Noun holders who missed the fork escrow window, cannot join the fork in the unreasonably small fork period and do no have sufficient voting power to fork again.

While the accidental setting of the lower bound to an undesirable value poses a lower risk than that of the upper bound, this is yet another vector of attack by a malicious majority on forking capability/effectiveness. While the majority can upgrade the DAO entirely at will to circumvent all such guardrails, we hypothesise that would get more/all attention by token holders than modification of governance/fork parameters whose risk/impact may not be apparent immediately to non-technical or even technical holders.

So unless there is an automated impact review/analysis performed as part of governance processes, such proposal vectors on governance/forking parameters should be considered as posing non-negligible risk.

Impact: Inattentive minority Noun holders are unable to join the fork and forced to stick with the original DAO.

Low likelihood + High impact = Medium severity.

Recommendation: (1) Consider a `MIN_FORK_PERIOD` value e.g. 2 days for `forkPeriod` which gives a better opportunity to inattentive/unsure minority Noun holders to join the fork now that it already exists. (2) Consider preventing the decrease of `forkPeriod` if a fork is in the escrow period.

Nouns:

Mitigation: PR 716.

Spearbit: Verified that PR 716 fixes the issue by enforcing a `MIN_FORK_PERIOD` of 2 days in `_setForkPeriod()`.

5.2.14 A malicious DAO can prevent forking by manipulating the `forkThresholdBPS` value

Severity: Medium Risk

Context: NounsDAOV3Admin.sol#L530-L537

Description: While some of the documentation, see 1 and 2, note that the fork threshold is expected to be 20%, the `forkThresholdBPS` is a DAO governance controlled value that may be modified via `_setForkThresholdBPS()`.

A malicious majority can prevent forking at any time by setting the `forkThresholdBPS` to an unreasonably high value that is >= majority voting power. For a fork that is slowly gathering support via escrowing (thus giving time for a DAO proposal to be executed) , a malicious majority can reactively manipulate `forkThresholdBPS` to prevent that fork from being executed.

While the governance process gives an opportunity to detect and block such malicious proposals, the assumption is that a malicious majority can force through any proposal, even a visibly malicious one. Also, it is not certain that all governance proposals undergo thorough scrutiny of security properties and their impacts. Token holders need to actively monitor all proposals for malicious updates to create, execute and join a fork before such a proposal takes effect.

A malicious majority can prevent a minority from forking by manipulating the `forkThresholdBPS` value.

Low likelihood + High impact = Medium severity.

Recommendation: Mitigation options: (1) Consider a `MAX_FORK_THRESHOLD` value e.g. 50% for `forkThresholdBPS`. (2) Consider preventing the increase of `forkThresholdBPS` if a fork is in the escrow period.

Nouns:

> We don't think a maximum value is needed here. If a malicious majority can set this value to be unreasonably high without the minority noticing, it's likely they would also be able to upgrade the DAO contracts without them noticing.

Spearbit: Acknowledged.

5.2.15 A malicious DAO can prevent/deter token holders from executing/joining a fork by including arbitrary addresses in `erc20TokensToIncludeInFork`

Severity: Medium Risk

Context: NounsDAOV3Fork.sol#L224-L228

Description: As motivated in the fork spec, forking is a minority protection mechanism that should always allow a group of minority token holders to exit together into a new instance of Nouns DAO.

However, a malicious majority in the original DAO may include arbitrary addresses in `erc20TokensToIncludeInFork` (modifiable via `_setErc20TokensToIncludeInFork()`) that revert on `balanceOf()` or `transfer()` calls to prevent token holders from executing or joining a fork. While the governance process gives an opportunity to detect and block such malicious proposals, the assumption is that a malicious majority can force through any proposal, even a visibly malicious one. Also, it is not certain that all governance proposals undergo thorough scrutiny of security properties which allows a proposal to hide malicious ERC20 tokens and get them included in the DAO's allow list. Token holders need to monitor all proposals for malicious updates to create, execute and join a fork before such a proposal takes effect.

Furthermore, a forking token holder may not necessarily want to receive all the DAO's ERC20 tokens in their new fork DAO for various reasons. For e.g., custody of certain ERC20 tokens may not be legal in their regulatory jurisdictions and so they may not want to interact with a DAO whose treasury holds such tokens and may send them at some point (e.g. rage quit). Minority token holders may even want to fork specifically because of an ERC20's presence or proposed inclusion in the DAO treasury.

Giving forking holders a choice of ERC20s to take to fork DAO gives them a choice to fork anytime only with ETH and a subset of approved tokens if the DAO has already managed to add malicious/contentious ERC20s in the list.

1. A malicious DAO can prevent unsuspecting/inattentive or future token holders from forking and taking out their pro rata funds, which is the main motivation for minority protection as specified.
2. A forking token holder is forced to end up with a fork DAO treasury that has all the original DAO's ERC20 tokens without having a choice, which may deter them from creating/executing/joining a fork in the first place.

Low likelihood + High impact = Medium severity.

Recommendation: Reconsider the design choice and threat/trust models to evaluate the feasibility of a better one, e.g. when a fork gets triggered, the first caller of `escrowToFork()` (the initiator) could get to choose a subset of `erc20TokensToIncludeInFork` for their fork, after which `executefork()` and all `joinFork()`s only send those tokens to the fork DAO treasury.

Nouns:

> Won't fix. We see the risk of adding a malicious token to `erc20TokensToIncludeInQuit` to be low, because it needs to go through a proposal process which is being reviewed by the DAO members. In addition, we don't want to give the first caller of `escrowToFork` special privileges that may not be in agreement with other members who wish to fork.

Spearbit: Acknowledged.

5.2.16 A malicious new DAO can prevent/deter token holders from rage quitting by including arbitrary addresses in `erc20TokensToIncludeInQuit`

Severity: Medium Risk

Context: NounsDAOLogicV1Fork.sol#L201-L215

Description: As described in the fork spec: "*New DAOs are deployed with vanilla ragequit in place; otherwise it's possible for a new DAO majority to collude to hurt a minority, and the minority wouldn't have any last resort if they can't reach the forking threshold; furthermore bullies/attackers can recursively chase minorities into fork DAOs in an undesired attrition war.*".

However, a malicious new DAO may include arbitrary addresses in `erc20TokensToIncludeInQuit` (modifiable via `_setErc20TokensToIncludeInQuit()`) that revert on `balanceOf()` or `transfer()` calls to prevent token holders from rage quitting. While the governance process gives an opportunity to detect and block such malicious proposals, the assumption is that a malicious majority can force through any proposal, even a visibly malicious one. Also, it is not certain that all governance proposals undergo thorough scrutiny of security properties which allows a proposal to hide malicious ERC20 tokens and get them included in the DAO's allow list. Token holders need to monitor all proposals for malicious updates and rage quit before such a proposal takes effect.

Furthermore, a rage quitting token holder may not necessarily want to receive all the DAO's ERC20 tokens for various reasons. For e.g., custody of certain ERC20 tokens may not be legal in their regulatory jurisdictions.

(1) A malicious new DAO can prevent unsuspecting/inattentive token holders from rage quitting and taking out their pro rata funds, which is a critical capability for minority protection as specified. (2) A rage quitting token holder is forced to receive all the DAO's ERC20 tokens without having a choice, which may deter them from quitting.

Recommendation: Consider accepting an ERC20 token list, which is a subset of `erc20TokensToIncludeInQuit`, from the user to allow them to choose/skip ERC20 tokens.

Nouns: Fix: PR 732.

Spearbit: Verified that PR 732 fixes the issue as recommended.

5.2.17 Missing check for vetoed proposal's target timelock can cancel transactions from other proposals on new DAO treasury

Severity: Medium Risk

Context: NounsDAOV3Proposals.sol#L527-L544 NounsDAOV3Proposals.sol#L435 NounsDAOV3Proposals.sol#L472 NounsDAOV3Proposals.sol#L590

Description: `veto()` always assumes that the proposal being vetoed is targeting `ds.timelock` (i.e. the new DAO treasury) instead of checking via `getProposalTimelock()` as done by `queue()`, `execute()` and `cancel()` functions. If the proposal being vetoed were targeting `timelockV1` (i.e. original DAO treasury) then this results in calling `cancelTransaction()` on the wrong timelock which sets `queuedTransactions[txHash]` to false for values of target, value, signature, data and eta.

The proposal state is vetoed with zombie queued transactions on `timelockV1` which will never get executed. But if there coincidentally were valid transactions with the same values (of target, value, signature, data and eta) from other proposals queued (assuming in the same block and that both timelocks have the same delay so that `eta` is the same) on `ds.timelock` then those would unexpectedly and incorrectly get dequeued and will not be executed even when these other `ds.timelock` targeting proposals were neither vetoed nor cancelled. Successfully voted proposals on new DAO treasury have their transactions cancelled before execution.

Confirmed with PoC: veto_poc.txt

Low likelihood + High impact = Medium severity.

Recommendation: Check vetoed proposal's target timelock via `getProposalTimelock()` and use that as done by `queue()`, `execute()` and `cancel()` functions.

Nouns: Mitigation: PR 715.

Spearbit: Verified that PR 715 fixes the issue as recommended.

5.2.18 Proposal threshold can be bypassed through the `proposeBySigs()` function

Severity: Medium Risk

Context: NounsDAOV3Proposals.sol#L229

Description: The function `proposeBySigs()` allows users to delegate their voting power to a proposer through signatures so the proposer can create a proposal. The only condition is that the sum of the signers voting power should be higher than the proposal threshold.

In the line `uint256 proposalId = ds.proposalCount = ds.proposalCount + 1;`, the `ds.proposalCount` is increased but the proposal has not been created yet, meaning that the `NounsDAOStorageV3.Proposal` struct is, at this point, uninitialized, so when the checkNoActiveProp() function is called the proposal state is `DEFEATED`.

As the proposal state is `DEFEATED` the `checkNoActiveProp()` call would not revert in the case that a signer is repeated in the `NounsDAOStorageV3.ProposerSignature[]` array:

```
function checkNoActiveProp(NounsDAOStorageV3.StorageV3 storage ds, address proposer) internal view {
    uint256 latestProposalId = ds.latestProposalIds[proposer];
    if (latestProposalId != 0) {
        NounsDAOStorageV3.ProposalState proposersLatestProposalState = state(ds, latestProposalId);
        if (
            proposersLatestProposalState == NounsDAOStorageV3.ProposalState.ObjectionPeriod ||
            proposersLatestProposalState == NounsDAOStorageV3.ProposalState.Active ||
            proposersLatestProposalState == NounsDAOStorageV3.ProposalState.Pending ||
            proposersLatestProposalState == NounsDAOStorageV3.ProposalState.Updatable
        ) revert ProposerAlreadyHasALiveProposal();
    }
}
```

Because of this it is possible to bypass the proposal threshold and create any proposal by signing multiple `proposerSignatures` with the same signer over and over again. This would keep increasing the total voting power accounted by the smart contract until this voting power is higher than the proposal threshold.

Medium likelihood + Medium Impact = Medium severity.

Recommendation: Consider creating the proposal before verifying the `NounsDAOStorageV3.ProposerSignature[]` array by calling `createNewProposal()` before `verifySignersCanBackThisProposalAndCountTheirVotes()` function. By doing this, when `verifySignersCanBackThisProposalAndCountTheirVotes()` is called the proposal state will be correct and any repeated signer would cause a revert during the `checkNoActiveProp(ds, signer);` call.

Nouns: Followed Spearbit's recommendation and fixed the issue in PR 711.

Spearbit: Acknowledged.

5.3 Low Risk

5.3.1 Attacker can utilize bear market conditions to profit from forking the Nouns DAO

Severity: Low Risk

Context: NounsDAOV3Fork.sol#L212-L231

Description: An economic attack vector has been identified that could potentially compromise the integrity of the Nouns DAO treasury, specifically due to the introduction of forking functionality. Currently, the treasury holds approximately `$24,745,610.99` in ETH and about `$27,600,000` in STETH. There are roughly `738` nouns tokens. As per OpenSea listings, the cheapest nouns-token can be purchased for about `31 ETH`, approximately `$53,000`. Meanwhile, the daily auction price for the nouns stands at approximately `28 ETH`, which equals about `$48,600`.

A prospective attacker may exploit the current bear market conditions, marked by discounted price, to buy multiple nouns-tokens at a low price, execute a fork to create a new DAO and subsequently claim a portion of the treasury. This act would result in the attacker gaining more than they invested at the expense of the Nouns DAO treasury.

To illustrate, if the forking threshold is established at `20%`, an attacker would need `148` nouns to execute a fork.

Consider the scenario where a user purchases `148` nouns for a total of `4588 ETH` (`148 x 31 ether`). The `forkTreasury.balance` would be `2679.27 ETH`, and the `contract_stETH.balanceOf(forkTreasury)` would stand at `3000.7 ETH`. The total ETH obtained would amount to `5680.01 ETH`, thereby yielding a profit of `1092 ETH` (`$2,024,568`).

Recommendation: Consider updating original DAO auction house reserve price and evaluate market controlling initiatives, say nouns buybacks with treasury funds and their subsequent burning (given low market prices it will be profit generating activity in terms of nouns valuation).

Nouns:

> We are aware of the incentives the fork mechanism creates. For now we will not introduce any more changes, but this is a topic that will continue to be discussed.

Spearbit: Acknowledged.

5.3.2 Setting NounsAuctionHouse's `timeBuffer` too big is possible, which will freeze bidder's funds

Severity: Low Risk

Context: NounsAuctionHouse.sol#L161-L169

Description: It's now possible to set `timeBuffer` to an arbitrary big value with setTimeBuffer(), there is no control:

- NounsAuctionHouse.sol#L161-L169

```
    /**
     * @notice Set the auction time buffer.
     * @dev Only callable by the owner.
     */
    function setTimeBuffer(uint256 _timeBuffer) external override onlyOwner {
        timeBuffer = _timeBuffer;

        emit AuctionTimeBufferUpdated(_timeBuffer);
    }
```

This can freeze user funds as NounsAuctionHouse holds current bid, but the its release in conditional to `block.timestamp >= _auction.endTime`:

- NounsAuctionHouse.sol#L96-L98

```
    function settleAuction() external override whenPaused nonReentrant {
        _settleAuction();
    }
```

- NounsAuctionHouse.sol#L221-L234

```
    function _settleAuction() internal {
        INounsAuctionHouse.Auction memory _auction = auction;

        require(_auction.startTime != 0, "Auction hasn't begun");
        require(!_auction.settled, 'Auction has already been settled');
>>      require(block.timestamp >= _auction.endTime, "Auction hasn't completed");

        auction.settled = true;

        if (_auction.bidder == address(0)) {
            nouns.burn(_auction.nounId);
        } else {
            nouns.transferFrom(address(this), _auction.bidder, _auction.nounId);
        }
```

Which can be set to be arbitrary big value, say `10^6 years`, effectively freezing current bidder's funds:

- NounsAuctionHouse.sol#L104-L129

```
    function createBid(uint256 nounId) external payable override nonReentrant {
        ...

        // Extend the auction if the bid was received within `timeBuffer` of the auction end time
        bool extended = _auction.endTime - block.timestamp < timeBuffer;
        if (extended) {
>>          auction.endTime = _auction.endTime = block.timestamp + timeBuffer;
        }
```

I.e. permissionless `settleAuction()` mechanics will be disabled.

Current bidder's funds will be frozen for an arbitrary time. As the new setting needs to pass voting, the probability is very low. In the same time it is higher for any forked DAO than for original one, so, while the issue is present in the V1 and V2, it becomes more severe in V3 in the context of the forked DAO. The impact is high, being long-term freeze of the bidder's native tokens.

Recommendation: Consider introducing upper limit to `timeBuffer`, say `1 day`.

Nouns:

> At this point, due to this requiring going through a DAO proposal, we'll leave it at won't fix

Spearbit: Acknowledged.

5.3.3 Veto renouncing in the original DAO or rage quit blocking in a forked DAO as a result of any future proposals will open up the way for 51% attacks

Severity: Low Risk

Context: NounsDAOLogicV2.sol#L900-L915, NounsDAOV3Admin.sol#L318-L333, NounsDAOLogicV1Fork.sol#L195-L222

Description: It is possible to renounce veto power in V1, V2 and V3 versions of the protocol or upgrade forked V1 to block or remove rage quit. While it is a part of standard workflow, these operations are irreversible and open up a possibility of all variations of `51% attack`. As a simplest example, in the absence of veto functionality a majority can introduce and execute a proposal to move all DAO treasury funds to an address they control. Also, there is a related vector, incentivized bad faith voting.

`_burnVetoPower()` exists in `V1`, `V2` and `V3`:

In `NounsDAOLogicV1` :

```
    /**
     * @notice Burns veto priviledges
     * @dev Vetoer function destroying veto power forever
     */
    function _burnVetoPower() public {
        // Check caller is pendingAdmin and pendingAdmin  address(0)
        require(msg.sender == vetoer, 'NounsDAO::_burnVetoPower: vetoer only');

        _setVetoer(address(0));
    }
```

In NounsDAOLogicV2:

```
    /**
     * @notice Burns veto priviledges
     * @dev Vetoer function destroying veto power forever
     */
    function _burnVetoPower() public {
        // Check caller is vetoer
        require(msg.sender == vetoer, 'NounsDAO::_burnVetoPower: vetoer only');

        // Update vetoer to 0x0
        emit NewVetoer(vetoer, address(0));
        vetoer = address(0);

        // Clear the pending value
        emit NewPendingVetoer(pendingVetoer, address(0));
        pendingVetoer = address(0);
    }
```

In NounsDAOLogicV3:

```
    /**
     * @notice Burns veto priviledges
     * @dev Vetoer function destroying veto power forever
     */
    function _burnVetoPower(NounsDAOStorageV3.StorageV3 storage ds) public {
        // Check caller is vetoer
        require(msg.sender == ds.vetoer, 'NounsDAO::_burnVetoPower: vetoer only');

        // Update vetoer to 0x0
        emit NewVetoer(ds.vetoer, address(0));
        ds.vetoer = address(0);

        // Clear the pending value
        emit NewPendingVetoer(ds.pendingVetoer, address(0));
        ds.pendingVetoer = address(0);
    }
```

Also, `veto()` was removed from NounsDAOLogicV1Fork, and the only mitigation to the same attack is rage `quit()`:

- NounsDAOLogicV1Fork.sol#L195-L222

```
    /**
     * @notice A function that allows token holders to quit the DAO, taking their pro rata funds,
     * and sending their tokens to the DAO treasury.
     * Will revert as long as not all tokens were claimed, and as long as the delayed governance has
     ↪  not expired.
     * @param tokenIds The token ids to quit with
     */
    function quit(uint256[] calldata tokenIds) external nonReentrant {
        checkGovernanceActive();

        uint256 totalSupply = adjustedTotalSupply();

        for (uint256 i = 0; i < tokenIds.length; i++) {
            nouns.transferFrom(msg.sender, address(timelock), tokenIds[i]);
        }

        for (uint256 i = 0; i < erc20TokensToIncludeInQuit.length; i++) {
            IERC20 erc20token = IERC20(erc20TokensToIncludeInQuit[i]);
            uint256 tokensToSend = (erc20token.balanceOf(address(timelock)) * tokenIds.length) /
            ↪  totalSupply;
            bool erc20Sent = timelock.sendERC20(msg.sender, address(erc20token), tokensToSend);
            if (!erc20Sent) revert QuitERC20TransferFailed();
        }

        uint256 ethToSend = (address(timelock).balance * tokenIds.length) / totalSupply;
        bool ethSent = timelock.sendETH(msg.sender, ethToSend);
        if (!ethSent) revert QuitETHTransferFailed();

        emit Quit(msg.sender, tokenIds);
    }
```

This means that any malfunction in this function, as an example if USDC is added to `erc20TokensToIncludeInQuit`, while a minority of forked nouns holders was previously black-listed by USDC contract, will open up the possibility of majority attack on them, i.e. there will be no way to stop any majority backed malicious proposal from affecting the DAO held funds of such holders.

Nouns holders that aren't aware enough of the importance of functioning `veto()` for original DAO and `quit()` for the forked DAO, can pass a proposal that renounce veto or [fully or partially] block `quit()`, enabling the `51% attack`. Such change will be irreversible and if a majority forms and acts before any similar mitigation functionalities be reinstalled, the whole DAO funds of the rest of the holders can be lost.

Per very low likelihood (which increases with the switch from `veto()` to `quit()` as a safeguard), and high funds loss impact, setting the severity to be low.

Recommendation: Key mitigation here is educating the holders of the roles of these two functions. The surface will not exist as long as most of the holders are aware of the possibility of the attack and these guards being in place and reject any proposal affecting them.

Consider highlighting `_burnVetoPower()` effect on opening `51% attack` vector for original DAOs, and emphasizing the *availability* of forked DAO `quit()` as the only safeguard in place against the same vector.

Note, that with the absence of `veto()` in the forked DAO it will be a wider range of possible changes that have a potential to create this vector (not only `quit()` removal, but `quit()` blocking for any reason).

Nouns:

> The DAO members are well aware of the risk of 51% attack and the importance of veto and fork/quit options. We will make sure this stays in awareness of all members.

Spearbit: Acknowledged.

5.3.4 The try-catch block at `NounsAuctionHouseFork` will only catch errors that contain strings

Severity: Low Risk

Context: NounsAuctionHouseFork.sol#L213-L229

Description: This issue has been previously identified and documented in the Nouns Builder Code4rena Audit.

The `catch Error(string memory)` within the `try/catch block` in the `_createAuction` function only catches reverts that include strings. At present, in the current version of the `NounsAuctionHouseFork` there are not reverts without a string.

But, given the fact that the `NounsAuctionHouseFork` and the `NounsTokenFork` contracts are meant to be upgradable, if a future upgrade in the `NounsTokenFork:mint()` replaces the require statements with custom errors, the existing catch statement won't be able to handle the reverts, potentially leading to a faulty state of the contract.

Here's an example illustrating that the `catch Error(string memory)` won't catch reverts with custom errors that don't contain strings:

```
contract Test1 {
    bool public error;
    Test2 test;

    constructor() {
         test = new Test2();
    }

    function testCustomErr() public{

        try test.revertWithRevert(){

        } catch Error(string memory) {
            error = true;
        }
    }

    function testRequire() public{

        try test.revertWithRequire(){

        } catch Error(string memory) {
            error = true;
        }
    }
}

contract Test2 {
    error Revert();
    function revertWithRevert() public{
        revert Revert();
    }

    function revertWithRequire() public {
        require(true == false, "a");
    }
}
```

Recommendation: If the `NounsTokenFork:mint()` methods of the `NounsTokenFork` changes to include custom errors as the revert reason, it is recommended to modify the `catch Error(string memory)` to also catch errors that don't contain strings.

However, it is not advisable to change `catch Error(string memory)` to `catch`, as this modification could unintentionally capture unwanted reverts, such as those due to an `out-of-gas` condition.

Nouns:

> We will take this into consideration if the errors change from strings to custom errors. For now, we won't fix.

Spearbit: Acknowledged.

5.3.5 Private keys are read from the `.env` environment variable in the deployment scripts

Severity: Low Risk

Context: DeployDAOV3NewContractsBase.s.sol#L53, ProposeENSReverseLookupConfigMainnet.s.sol#L18, DeployDAOV3DataContractsBase.s.sol#L21, ProposeDAOV3UpgradeMainnet.s.sol#L24, ProposeDAOV3UpgradeTestnet.s.sol#L32, ProposeTimelockMigrationCleanupMainnet.s.sol#L22

Description: It has been identified that the private key of privileged (`PROPOSER_KEY` and `DEPLOYER_PRIVATE_KEY`) accounts are read the environment variables within scripts.

The deployer address is verified on Etherscan as `Nouns DAO: Deployer`. Additionally, since the proposal is made by the account that owns the `PROPOSER_KEY`, it can be assumed that the proposer owns at least some Nouns.

Given the privileged status of the deployer and the proposer, unauthorized access to this private key could have a negative impact in the reputation of the Nouns DAO.

The present method of managing private keys, i.e., through environment variables, represents a potential security risk. This is due to the fact that any program or script with access to the process environment can read these variables.

As mentioned in the Foundry documentation:

> This loads in the private key from our .env file. Note: you must be careful when exposing private keys in a .env file and loading them into programs. This is only recommended for use with non-privileged deployers or for local / test setups. For production setups please review the various wallet options that Foundry supports.

Recommendation: Consider employing alternative methods to load the private keys in the deployment scripts. Some alternatives as suggested in the `ETHSecurity channel` include:

1. `AWS Secret Manager`
2. `AWS CloudHSM`
3. `AWS KMS External Key Store (XKS)`
4. `AWS Nitro Enclaves`

Nouns:

> The `PROPOSER_KEY` doesn't need to necessarily own Nouns, it can be delegated some voting power. In any case, we will consider changing this, but for now, won't fix.

Spearbit: Acknowledged.

5.3.6 Objection period will be disabled after the update to V3 is completed

Severity: Low Risk

Context: ProposeDAOV3UpgradeMainnet.s.sol, ProposeTimelockMigrationCleanupMainnet.s.sol

Description: Nouns DAO V3 introduces a new functionality called objection-only period. This is a conditional voting period that gets activated upon a last-minute proposal swing from defeated to successful, affording against voters more reaction time. Only against votes will be possible during the objection period.

After the proposals created in `ProposeDAOV3UpgradeMainnet.s.sol` and `ProposeTimelockMigrationCleanupMainnet.s.sol` are executed `lastMinuteWindowInBlocks` and `objectionPeriodDurationInBlocks` will still remain set to 0. A new proposal will have to be created, passed and executed in the DAO that calls the `_setLastMinuteWindowInBlocks()` and `_setObjectionPeriodDurationInBlocks()` functions to enable this functionality.

Recommendation: Consider adding to the `ProposeTimelockMigrationCleanupMainnet.s.sol`'s proposal 2 new transactions that call and set the `_setLastMinuteWindowInBlocks()` and `_setObjectionPeriodDurationInBlocks()`.

Nouns:

> This was done intentionally as we want the upgrade to start with this feature turned off. We want the DAO to vote separately on the parameters they want to turn it on with. Won't fix.

Spearbit: Acknowledged.

5.3.7 Potential risks from outdated OpenZeppelin dependencies in the Nouns DAO v3

Severity: Low Risk

Context: yarn.lock#L4191-L4192, yarn.lock#L4196-L4197

Description: The OpenZeppelion libraries are being used throughout the Nouns DAO v3 codebase. These libraries however, are locked at version `4.4.0`, which is an outdated version that has some known vulnerabilities.

Specifically:

- The `SignatureChecker.isValidSignatureNow` is not expected to revert. However, an incorrect assumption about `Solidity 0.8's abi.decode` allows some cases to revert, given a target contract that doesn't implement `EIP-1271` as expected. The contracts that may be affected are those that use `SignatureChecker` to check the validity of a signature and handle invalid signatures in a way other than reverting.
- The `ERC165Checker.supportsInterface` is designed to always successfully return a boolean, and under no circumstance revert. However, an incorrect assumption about `Solidity 0.8's abi.decode` allows some cases to revert, given a target contract that doesn't implement `EIP-165` as expected, specifically if it returns a value other than 0 or 1. The contracts that may be affected are those that use `ERC165Checker` to check for support for an interface and then handle the lack of support in a way other than reverting.

At present, these vulnerabilities do not appear to have an impact in the Nouns DAO codebase, as corresponding functions revert upon failure. Nevertheless, these vulnerabilities could potentially impact future versions of the codebase.

Recommendation: As a security measure, it is advisable to update the version of the `OpenZeppelin` libraries in both the `yarn.lock` and the `package.json` files.

Nouns:

> We will consider it for the future. For now, we won't fix.

Spearbit: Acknowledged.

5.3.8 DAO withdraws forked ids from escrow without emphasizing total supply increase which contradicts the spec and can catch holders unaware

Severity: Low Risk

Context: NounsDAOV3Fork.sol#L160-L178

Description: Withdrawal original nouns with ids of the forked tokens from escrow after the successful fork is a material event for all original nouns holders as the adjusted total supply is increased as long as withdrawal recipient is not treasury.

There were special considerations regarding Nouns withdrawal impact after the fork:

```
For this reason we're considering a change to make sure transfers go through a new function that helps
↪  Nouners
understand the implication, e.g. by setting the function name to withdrawNounsAndGrowTotalSupply
or something similar, as well as emitting events that indicate the new (and greater) total supply used
↪  by the DAO.
```

However, currently `withdrawDAONounsFromEscrow()` neither have a special name, nor mentions the increase of adjusted total supply when `to != ds.timelock`:

- NounsDAOV3Fork.sol#L160-L178

```
    /**
     * @notice Withdraws nouns from the fork escrow after the fork has been executed
     * @dev Only the DAO can call this function
     * @param tokenIds the tokenIds to withdraw
     * @param to the address to send the nouns to
     */
    function withdrawDAONounsFromEscrow(
        NounsDAOStorageV3.StorageV3 storage ds,
        uint256[] calldata tokenIds,
        address to
    ) external {
        if (msg.sender != ds.admin) {
            revert AdminOnly();
        }

        ds.forkEscrow.withdrawTokens(tokenIds, to);

        emit DAOWithdrawNounsFromEscrow(tokenIds, to);
    }
```

Nouns holder might not understand the consequences of withdrawing the nouns from escrow and support such a proposal, while as of now it is approximately `USD 65k` loss per noun withdrawn cumulatively for current holders.

The vulnerability scenario here is a holder supporting the proposal without understanding the consequences for supply, as no emphasis is made, and then suffers their share of loss as a result of its execution.

Per low likelihood and impact setting the severity to be low.

Recommendation: Consider emitting a special event when `to != timelock`, for example:

```
    if (to != address(ds.timelock)) {
        emit DAONounsSupplyIncreasedFromEscrow(tokenIds.length, to);
    }
```

In order to emphasize the supply increase consider splitting `withdrawDAONounsFromEscrow()` into 2 functions:

- First, for example, `withdrawDAONounsFromEscrowToTreasury`, having no `to` argument and performing `ds.forkEscrow.withdrawTokens(tokenIds, address(ds.timelock))`

- Second, for example, `withdrawDAONounsFromEscrowIncreasingTotalSupply`, requiring `to != address(ds.timelock)`, performing `ds.forkEscrow.withdrawTokens(tokenIds, to)` and emitting `DAONounsSupplyIncreasedFromEscrow`

Nouns:

Fix: nounsDAO/nouns-monorepo#735.

Spearbit: Fix looks good.

5.3.9 USDC-paying proposals executing between `ProposeDAOV3UpgradeMainnet` and `ProposeTimelockMigrationCleanupMainnet` will fail

Severity: Low Risk

Context: Known-Issues ProposeDAOV3UpgradeMainnet.s.sol#L106-L116

Description: As explained in one of the Known Issues, `ProposeDAOV3UpgradeMainnet` contains a proposal that transfers the ownership of `PAYER_MAINNET` and `TOKEN_BUYER_MAINNET` from `timelockV1` to `timelockV2`. There could be older USDC-paying proposals executing after `ProposeDAOV3UpgradeMainnet` which assume `timelockV1` ownership of these contracts.

Older USDC-paying proposals executing after `ProposeDAOV3UpgradeMainnet` will fail.

Recommendation: While the proposed mitigation is: "*Our mitigation to this issue will be communication with the DAO ahead of time, and should we see any proposals in this state, we will contact their proposers immediately and help them re-propose.*", the number of affected proposals could be reduced if the ownership transfers of `PAYER_MAINNET` and `TOKEN_BUYER_MAINNET` are moved from `ProposeDAOV3UpgradeMainnet` to `ProposeTimelockMigrationCleanupMainnet`, which will at least prevent the older proposals executing between these two proposals from failing due to this reason.

Nouns:

Fixed: nounsDAO/nouns-monorepo#734.

Spearbit: Verified that PR 734 fixes the issue as recommended.

5.3.10 Zero value ERC-20 transfers can be performed on sending treasury funds to quitting member or forked DAO, denying the whole operation if one of `erc20TokensToIncludeInQuit` tokens doesn't allow this

Severity: Low Risk

Context: NounsDAOLogicV1Fork.sol#L210-L215, NounsDAOV3Fork.sol#L225-L230

Description: Some tokens do not allow for zero value transfers. Such behaviour do not violate ERC-20 standard, is not anyhow prohibited and can occur in any non-malicious token.

As a somewhat well-known example Aave's `LEND` requires amount to be positive:

- etherscan.io/address/0x80fb784b7ed66730e8b1dbd9820afd29931aab03#code#L74

```
function transfer(address _to, uint256 _value) returns(bool) {
    require(balances[msg.sender] >= _value);
    require(balances[_to] + _value > balances[_to]);
```

As `stETH`, which is currently used by Nouns treasury, is upgradable, it cannot be ruled out that it might be requiring the same in the future for any reason.

- etherscan.io/token/0xae7ab96520de3a18e5e111b5eaab095312d7fe84#code

Zero value itself can occur in a situation when valid token was added to `erc20TokensToIncludeInFork`, but this token timelock balance is currently empty.

NounsDAOLogicV1Fork's `quit()` and NounsDAOV3Fork's `executeFork()` and `joinFork()` will be unavailable in such scenario, i.e. the DAO forking workflow will be disabled.

Since the update of `erc20TokensToIncludeInFork` goes through proposal mechanics and major tokens rarely upgrade, while there is an additional requirement of empty balance, the cumulative probability of the scenario can be deemed quite low, while the core functionality blocking impact is high, so setting the severity to be low.

Recommendation: Consider controlling amount to be positive in both cases, for example:

NounsDAOLogicV1Fork's `quit()`:

```
        for (uint256 i = 0; i < erc20TokensToIncludeInQuit.length; i++) {
            IERC20 erc20token = IERC20(erc20TokensToIncludeInQuit[i]);
            uint256 tokensToSend = (erc20token.balanceOf(address(timelock)) * tokenIds.length) /
            ↪  totalSupply;
+           if (tokensToSend > 0) {
                bool erc20Sent = timelock.sendERC20(msg.sender, address(erc20token), tokensToSend);
                if (!erc20Sent) revert QuitERC20TransferFailed();
+           }
        }
```

The `NounsDAOV3Fork:sendProRataTreasury()` which is called during the `executeFork()` & `joinFork()` functions:

```
        for (uint256 i = 0; i < erc20Count; ++i) {
            IERC20 erc20token = IERC20(ds.erc20TokensToIncludeInFork[i]);
            uint256 tokensToSend = (erc20token.balanceOf(address(timelock)) * tokenCount) / totalSupply;
+           if (tokensToSend > 0) {
                bool erc20Sent = timelock.sendERC20(newDAOTreasury, address(erc20token), tokensToSend);
                if (!erc20Sent) revert ERC20TransferFailed();
+           }
        }
```

The motivation for the suggestion is also gas optimization: while DAO transfers are once per fork, in the `NounsDAOLogicV1Fork's quit()` case every quitting member will pay for such empty transfer, while its gas costs are substantial enough.

Nouns:

> Acknowledged. Fix PR here: nounsDAO/nouns-monorepo#727.

Spearbit: Fixed as recommended.

5.3.11 A signer of multiple proposals will cause all of them except one to fail creation

Severity: Low Risk

Context: NounsDAOV3Proposals.sol#L818 NounsDAOV3Proposals.sol#L787-L798

Description: Like proposers, signers are also allowed to back only one proposal at a time. As commented: "*This is a spam protection mechanism to limit the number of proposals each noun can back.*" However, unlike proposers who know which of their proposals are active and when, signers may not readily have that insight and can sign multiple proposals they may want to back. If more than one such proposal is proposed then only the first one will pass the `checkNoActiveProp()` for this signer and all the others will fail this check and thereby the proposal creation itself.

A signer of multiple proposals will cause all of them except one to fail creation. Other proposals will have to then exclude such signatures and resubmit. This could be accidental or used by malicious signers for griefing proposal creations.

Recommendation: Instead of reverting in `checkNoActiveProp()` when proposal signers already have an active proposal, have it return a boolean to help skip such signers from consideration in its callers. Not all signers included in `proposeBySigs()` have to necessarily be part of the proposal when created. `propose()` can choose to revert if `checkNoActiveProp()` returns false.

Nouns:

> Thanks for reporting. We agree this can improve UX a bit, but we think the clients should be able to identify signers with an active proposal. Won't fix

Spearbit: Acknowledged.

5.3.12 Single-step ownership change is risky

Severity: Low Risk

Context: NounsAuctionHouseFork.sol#L34, NounsTokenFork.sol#L20

Description: The codebase primarily follows a two-step ownership change pattern. However, in specific sections, a single-step ownership change is utilized.

Two-step ownership change is preferable, where:

- The current owner proposes a new address for the ownership change.
- In a separate transaction, the proposed new address can then claim the ownership.

Recommendation: Consider implementing a two-step ownership transfer process throughout the codebase, similar to the approach used in `Ownable2StepUpgradeable`.

Nouns:

> Since these ownership changes would have to go through a proposal process, this seems very low risk. We won't introduce a fix for this one.

Spearbit: Acknowledged.

5.3.13 No storage gaps for upgradeable contracts might lead to storage slot collision

Severity: Low Risk

Context: NounsDAOExecutorV2.sol#L46, NounsAuctionHouseFork.sol#L41, NounsDAOLogicV1Fork.sol#L105, NounsTokenFork.sol#L39

Description: When implementing upgradable contracts that inherit it is important that there are storage gaps, in case new storage variables are later added to the inherited contracts. If a storage gap variable isn't added, when the upgradable contract introduces new variables, it may override the variables in the inheriting contract.

As noted in the OpenZeppelin Documentation:

> You may notice that every contract includes a state variable named `__gap`. This is empty reserved space in storage that is put in place in Upgrade Safe contracts. It allows us to freely add new state variables in the future without compromising the storage compatibility with existing deployments. It isn't safe to simply add a state variable because it "shifts down" all of the state variables below in the inheritance chain.

Recommendation: To ensure the stability of future storage layout changes to the base contract, it is recommended to add a `__gap` variable as the last storage variable in these upgradeable contracts, resulting in a fixed total number (50 for example) of storage slots. Please note that as subsequent versions include additional storage variables, the `__gap` variable space will need to be adjusted accordingly to maintain the fixed number of slots.

Nouns:

> IIUC a storage gap is only helpful if the contract which has a storage gap is itself being inherited. The contracts you mentioned, NounsDAOExecutorV2, NounsAuctionHouseFork, NounsDAOLogicV1Fork and NounsTokenFork are not being inherited.
>
> Unless there's a larger reason, won't fix

Spearbit: Acknowledged.

5.3.14 The `version` string is missing from the domain separator allowing submission of signatures in different protocol versions

Severity: Low Risk

Context: NounsDAOV3Votes.sol#L164-L167, NounsDAOV3Proposals.sol#L981-L983, NounsDAOLogicV1Fork.sol#L560-L562, ERC721CheckpointableUpgradeable.sol#L146

Description: The `version` string seems to be missing from the domain separator.

According to EIP-712:

> Protocol designers only need to include the fields that make sense for their signing domain. Unused fields are left out of the struct type

While it's not a mandatory field as per the EIP-712 standard, it would be sensible for the protocol to include the version string in the domain separator, considering that the contracts are upgradable. For instance, if a user generates a signature for `version v1.0`, they may not want the signature to remain valid following an upgrade.

Recommendation: It is recommended to also include the `version` string in the domain separator.

Nouns:

> Due to very low risk and non trivial fix required, we will not fix this for now.

Spearbit: Acknowledged.

5.3.15 Two/three forks in a row will force expiration of execution-awaiting proposals

Severity: Low Risk

Context: ProposeDAOV3UpgradeMainnet.s.sol#L15 NounsDAOV3Admin.sol#L133 NounsDAOExecutorV2.sol#L80

Description: Proposal execution on the original DAO is disallowed during the forking period. While the proposed fork period is currently 7 days, `MAX_FORK_PERIOD` is 14 days. `GRACE_PERIOD`, which is the time allowed for a queued proposal to execute, has been increased from the existing 14 days to 21 days specifically to account for the fork period.

However, if there are three consecutive forks whose active fork periods add up to 21 days, or two forks in the worse case if the fork period is set to `MAX_FORK_PERIOD` then all queued proposals will expire and cannot be executed.

Malicious griefing forkers can collude to time and break-up their voting power to fork consecutively to prevent execution of queued proposal on the original DAO, thus forcing them to expire.

Recommendation: Consider introducing a short delay, e.g. 2-3 days, between the end of a fork and start of the next one. This will allow execution of queued proposals during this fork delay period without forcing their expiration.

Nouns:

> In order to perform such a griefing attack, given a threshold of 20% to fork, the attacker would need to have 36% (20% + 16%) of nouns to do 2 forks in a row, or 48% to do 3 forks in a row. This seems highly unlikely, therefore we won't be implementing a fix for this.

Spearbit: Acknowledged.

5.3.16 Withdrawing from fork escrow can be front-run to prevent withdrawal and force join the fork

Severity: Low Risk

Context: NounsDAOV3Fork.sol#L93C74-L100 NounsDAOV3Fork.sol#L109-L130

Description: `withdrawFromForkEscrow()` is meant to allow a fork escrowed holder change their mind about joining the fork by withdrawing their escrowed tokens. However, the current design allows another fork joining holder to front-run a `withdrawFromForkEscrow()` transaction with their `escrowToFork()` to exceed the fork threshold and also call `executeFork()` with it (if the threshold was already met then this doesn't even have to be another fork joining holder). This will cause `withdrawFromForkEscrow()` to fail because the fork period is now active and that holder is forced to join the fork with their previously escrowed tokens.

Scenario: Alice and Bob decide to create/join a fork with their 10 & 15 tokens respectively to meet the 20% fork threshold (assume 100 Nouns). Alice escrows first but then changes her mind and calls `withdrawFromForkEscrow()`. Bob observes this transaction (assume no private mempool) and front-runs it with his `escrowToFork()` + `executeFork()`. This forces Alice to join the fork instead of staying back.

`withdrawFromForkEscrow()` does not always succeed and is likely effective only in the early stages of escrow period but not towards the end when the fork threshold is almost met. Late fork escrowers do not have as much an opportunity as others to change their mind about joining the fork.

Recommendation: Front-running is always possible in such protocols and the standard mitigation is to use MEV protections for sensitive actions. Consider a fork execution delay of 1-2 days which is enforced after reaching the threshold but before executing the fork, so that escrowed participants can evaluate their decision to join the fork one last time.

Nouns:

> We don't think this is an issue. Escrowing your Noun to fork comes with the knowledge that it may fork. We don't need a "risk free" escrowing. Won't fix

Spearbit: Acknowledged.

5.3.17 A malicious proposer can replay signatures to create duplicate proposals

Severity: Low Risk

Context: NounsDAOV3Proposals.sol#L844-L851

Description: Suppose Bob and Alice sign a proposal for Carl, authorizing a transfer of exactly 100,000 USDC to a specified address (`xyz`) and their signatures were created with a long expiration time. Following the normal procedure, Carl creates the proposal, the vote is held, and the proposal enters the 'succeeded' state.

However, since Bob and Alice's signatures are still valid due to the long expiration time, Carl could reuse these signatures to create another proposal for an additional transfer of 100,000 USDC to the same `xyz` address, as long as Bob and Alice still retain their voting power/nouns. Thus, Carl could double the intended transfer amount without their explicit authorization.

While it is true that Bob and Alice can intervene by either cancelling the new proposal or invalidating their signatures before the creation of the second proposal, it necessitates them to take action, which may not always be feasible or timely.

Recommendation: Consider adding a nonce as part of these signatures and invalidate the nonce once is used.

Nouns:

> Adding a nonce system is too big of a change. We consider this issue to be very low risk, hence won't fix

Spearbit: Acknowledged.

5.3.18 Potential re-minting of previously burnt `NounsTokenFork`

Severity: Low Risk

Context: NounsTokenFork.sol#L142-L157, NounsDAOForkEscrow.sol#L168-L176

Description: In the current implementation of the `NounsTokenFork` contract, there is a potential vulnerability that allows for a previously burned `NounsTokenFork` to be re-minted. This risk occurs due to the user retaining the status of `escrow.ownerOfEscrowedToken(forkId, nounId)` even after the `claimFromEscrow()` function call.

Presently, no tokens are burned outside of the `NounsAuctionHouseFork` and are only burned in the case that no bids are placed for that `nounId`. However, this issue could become exploitable under the following circumstances:

1. If a new `burn()` functionality is added elsewhere in the code.
2. If a new contract is granted the `Minter` role.
3. If the `NounsAuctionHouseFork` is updated to a malicious implementation.

Additionally, exploiting this potential issue would lead to the `remainingTokensToClaim` variable decreasing, causing it to underflow `(<0)`. In this situation, some legitimate users would be unable to claim their tokens due to this underflow.

Recommendation: Consider updating the `escrowedTokensByForkId[forkId_][tokenId]` mapping in the `NounsDAOForkEscrow` contract every time `claimFromEscrow()` is called.

Nouns:

> Since there is no way to currently get the Noun burned without changing minter/contracts, we choose not to introduce new logic for fixing this. If the forked DAO chooses to enable burning, they could update the token contract and change the `claimFromEscrow()` logic.

Spearbit: Acknowledged.

5.3.19 A single invalid/expired/cancelled signature will prevent the creation and updation of proposals

Severity: Low Risk

Context: NounsDAOV3Proposals.sol#L956-L962 NounsDAOV3Proposals.sol#L815 NounsDAOV3Proposals.sol#L401

Description: For proposals created via `proposeBySigs()` or updated via `updateProposalBySigs()`, if the proposer includes even a single invalid/expired/cancelled signature (without performing offchain checks to prevent this scenario), `verifyProposalSignature()` will revert and the creation/updation of proposals will fail.

A proposer accidentally including one or more invalid/expired/cancelled signatures submitted accidentally by a signer will cause the proposal creation/updation to fail, lose gas used and will have to resubmit after checking and excluding such signatures. This also allows griefing by signers who intentionally submit an invalid/expired signature or a valid one which is later cancelled (using `cancelSig()`) just before the proposal is created/updated. Note that while the signers currently have cancellation powers which gives them a greater griefing opportunity even at later proposal states, that has been reported separately in a different issue.

Recommendation:

1. Consider having `verifyProposalSignature()` return a status code (invalid/expired/cancelled) which is then used to skip the consideration of such signatures. The voting power of other signatures may still add up to being sufficient for the proposal to be created or updated.
2. Consider checking that voting power of valid signatures during updation exceeds threshold instead of requiring the exact ordered list of original signers. The updation signers may be a subset of the original signers but still pass the proposal threshold to indicate support for updation.

Nouns:

> Acknowledged, but due to low risk, won't fix.

Spearbit: Acknowledged.

5.3.20 Missing require checks in `NounsDAOV3Proposals.execute()` and `executeOnTimelockV1()` functions

Severity: Low Risk

Context: NounsDAOV3Proposals.sol#L470, NounsDAOV3Proposals.sol#L481

Description: The following `require` checks are missing:

- Function`NounsDAOV3Proposals.execute()`:

```
require(proposal.executeOnTimelockV1 == false, 'NounsDAO::execute: executeOnTimelockV1 = true');
```

- Function `NounsDAOV3Proposals.executeOnTimelockV1()`:

```
require(proposal.executeOnTimelockV1 == true, 'NounsDAO::executeOnTimelockV1: executeOnTimelockV1 =
↪  false');
```

Due to the absence of these `require` checks, the `NounsDAOLogicV3` contract leaves open a vulnerability where if two identical proposals, with the exact same transactions, are concurrently queued in both the `timelockV1` and `timelock` contracts, the proposal originally intended for execution on `timelock` can be executed on `timelockV1` and vice versa.

The consequence of this scenario is that it essentially blocks or causes a Denial of Service to the legitimate execution path of the corresponding proposal for either `timelockV1` or `timelock`. This occurs because each proposal has been inadvertently executed on the unintended timelock contract due to the lack of a condition check that would otherwise ensure the correct execution path.

Recommendation: Consider adding a `require` that checks the `executeOnTimelockV1` flag in both functions.

Nouns:

The risk presented is very unlikely. Hence, leaving as is, won't fix.

Spearbit: Acknowledged.

5.3.21 Due to misaligned DAO and Executors logic any proposal will be blocked from execution at 'eta + GRACE_PERIOD' timestamp

Severity: Low Risk

Context: NounsDAOLogicV2.sol#L455-L473, NounsDAOV3Proposals.sol#L625-L652, NounsDAOLogicV1Fork.sol#L477-L493

Description: There is an inconsistency in treatment of the `eta + GRACE_PERIOD` moment of time in the proposal lifecycle: any proposal is executable in timelock at this timestamp, but have expired status in the DAO logic.

Both Executors do allow the executions when `block.timestamp == eta + GRACE_PERIOD`:

The `NounsDAOExecutor:executeTransaction()` function:

```
    function executeTransaction(
        address target,
        uint256 value,
        string memory signature,
        bytes memory data,
        uint256 eta
    ) public returns (bytes memory) {
        ...
        require(
            getBlockTimestamp() >= eta,
            "NounsDAOExecutor::executeTransaction: Transaction hasn't surpassed time lock."
        );
        require(
            getBlockTimestamp() <= eta + GRACE_PERIOD,
            'NounsDAOExecutor::executeTransaction: Transaction is stale.'
        );
```

The `NounsDAOExecutorV2:executeTransaction()` function:

```
    function executeTransaction(
        address target,
        uint256 value,
        string memory signature,
        bytes memory data,
        uint256 eta
    ) public returns (bytes memory) {
        ...
        require(
            getBlockTimestamp() >= eta,
            "NounsDAOExecutor::executeTransaction: Transaction hasn't surpassed time lock."
        );
        require(
            getBlockTimestamp() <= eta + GRACE_PERIOD,
            'NounsDAOExecutor::executeTransaction: Transaction is stale.'
        );
```

While all (V1,2, 3, and V1Fork) DAO state functions produce the expired state.

The `NounsDAOLogicV2:state()` function:

```
    function state(uint256 proposalId) public view returns (ProposalState) {
        require(proposalCount >= proposalId, 'NounsDAO::state: invalid proposal id');
        Proposal storage proposal = _proposals[proposalId];
        if (proposal.vetoed) {
            return ProposalState.Vetoed;
        } else if (proposal.canceled) {
            return ProposalState.Canceled;
        } else if (block.number <= proposal.startBlock) {
            return ProposalState.Pending;
        } else if (block.number <= proposal.endBlock) {
            return ProposalState.Active;
        } else if (proposal.forVotes <= proposal.againstVotes || proposal.forVotes <
        ↪  quorumVotes(proposal.id)) {
            return ProposalState.Defeated;
        } else if (proposal.eta == 0) {
            return ProposalState.Succeeded;
        } else if (proposal.executed) {
            return ProposalState.Executed;
>>      } else if (block.timestamp >= proposal.eta + timelock.GRACE_PERIOD()) {
            return ProposalState.Expired;
```

The `NounsDAOV3Proposals:stateInternal()` function:

```
    function stateInternal(NounsDAOStorageV3.StorageV3 storage ds, uint256 proposalId)
        internal
        view
        returns (NounsDAOStorageV3.ProposalState)
    {
        require(ds.proposalCount >= proposalId, 'NounsDAO::state: invalid proposal id');
        NounsDAOStorageV3.Proposal storage proposal = ds._proposals[proposalId];

        if (proposal.vetoed) {
            return NounsDAOStorageV3.ProposalState.Vetoed;
        } else if (proposal.canceled) {
            return NounsDAOStorageV3.ProposalState.Canceled;
        } else if (block.number <= proposal.updatePeriodEndBlock) {
            return NounsDAOStorageV3.ProposalState.Updatable;
        } else if (block.number <= proposal.startBlock) {
            return NounsDAOStorageV3.ProposalState.Pending;
        } else if (block.number <= proposal.endBlock) {
            return NounsDAOStorageV3.ProposalState.Active;
        } else if (block.number <= proposal.objectionPeriodEndBlock) {
            return NounsDAOStorageV3.ProposalState.ObjectionPeriod;
        } else if (isDefeated(ds, proposal)) {
            return NounsDAOStorageV3.ProposalState.Defeated;
        } else if (proposal.eta == 0) {
            return NounsDAOStorageV3.ProposalState.Succeeded;
        } else if (proposal.executed) {
            return NounsDAOStorageV3.ProposalState.Executed;
>>      } else if (block.timestamp >= proposal.eta + getProposalTimelock(ds, proposal).GRACE_PERIOD()) {
            return NounsDAOStorageV3.ProposalState.Expired;
```

The `NounsDAOLogicV1Fork:state()` function:

```
    function state(uint256 proposalId) public view returns (ProposalState) {
        require(proposalCount >= proposalId, 'NounsDAO::state: invalid proposal id');
        Proposal storage proposal = _proposals[proposalId];
        if (proposal.canceled) {
            return ProposalState.Canceled;
        } else if (block.number <= proposal.startBlock) {
            return ProposalState.Pending;
        } else if (block.number <= proposal.endBlock) {
            return ProposalState.Active;
        } else if (proposal.forVotes <= proposal.againstVotes || proposal.forVotes <
        ↪  proposal.quorumVotes) {
            return ProposalState.Defeated;
        } else if (proposal.eta == 0) {
            return ProposalState.Succeeded;
        } else if (proposal.executed) {
            return ProposalState.Executed;
>>      } else if (block.timestamp >= proposal.eta + timelock.GRACE_PERIOD()) {
            return ProposalState.Expired;
```

Impact:

Since both timelocks require sender to be admin, the valid proposal will be blocked from execution and forced to be expired when execution call time happens to be `proposal.eta + getProposalTimelock(ds, proposal).GRACE_PERIOD()`.

The probability of this exact timestamp to be reached is low, while the impact of successful proposal to be rendered invalid by itself is high. However, since there is enough time prior to that moment both for cancellation and execution and all these actions come through permissioned workflow the impact is better described as medium, so per low probability and medium impact setting the severity to be low.

Recommendation: As `eta` timestamp is included, consider the `if (block.timestamp > proposal.eta + GRACE_PERIOD) then {return Expired;}` kind of logic in all state functions:

At `NounsDAOLogicV2:state()`:

```
    function state(uint256 proposalId) public view returns (ProposalState) {
        ...
            return ProposalState.Executed;
-       } else if (block.timestamp >= proposal.eta + timelock.GRACE_PERIOD()) {
+       } else if (block.timestamp > proposal.eta + timelock.GRACE_PERIOD()) {
            return ProposalState.Expired;
```

At `NounsDAOV3Proposals:stateInternal()`:

```
    function stateInternal(NounsDAOStorageV3.StorageV3 storage ds, uint256 proposalId)
        internal
        view
        returns (NounsDAOStorageV3.ProposalState)
    {
        ...
            return NounsDAOStorageV3.ProposalState.Executed;
-       } else if (block.timestamp >= proposal.eta + getProposalTimelock(ds, proposal).GRACE_PERIOD()) {
+       } else if (block.timestamp > proposal.eta + getProposalTimelock(ds, proposal).GRACE_PERIOD()) {
            return NounsDAOStorageV3.ProposalState.Expired;
```

At `NounsDAOLogicV1Fork:state()`:

```
    function state(uint256 proposalId) public view returns (ProposalState) {
        ...
            return ProposalState.Executed;
-       } else if (block.timestamp >= proposal.eta + timelock.GRACE_PERIOD()) {
+       } else if (block.timestamp > proposal.eta + timelock.GRACE_PERIOD()) {
            return ProposalState.Expired;
```

Nouns:

> We consider this issue to be very low severity since it effectively just shortens the time a proposal can be queued by 1 second. We will not fix it at this time

Spearbit: In addition to that the impact also is that at `eta + timelock.GRACE_PERIOD` timestamp any proposal will not be cancellable, but this proposal transactions will be executable by a direct call to NounsDAOExecutorV2's `executeTransaction()`. Acknowledged.

5.3.22 A malicious DAO can increase the odds of proposal defeat by setting a very high value of `lastMinuteWindowInBlocks`

Severity: Low Risk

Context: NounsDAOV3Admin.sol#L219-L227 NounsDAOV3Votes.sol#L229-L257

Description: The goal of objection-only period, as documented, is to protect the DAO from executing proposals, that the majority would not want to execute. However, a malicious majority can abuse this feature by setting a very high value of `lastMinuteWindowInBlocks` (setter does not enforce max threshold), i.e. something very close to the voting period, to increase the probability of triggering objection-only period.

Example scenario: If `votingPeriod` = 2 weeks and a governance proposal somehow passed to set `lastMinuteWindowInBlocks` to a value very close to 100800 blocks i.e. ~2 weeks, then every proposal may end up with an objection-only period.

Impact: Every proposal may end up with an objection-only period which may not be required/expected.

Low likelihood + Low impact = Low severity.

Recommendation: Consider implementing a max threshold for `lastMinuteWindowInBlocks` preferably as a percentage value, say 70%, of the `votingPeriod` which would make any triggering of the objection period meaningful. This is because, switching of the proposal state from defeated to successful in the initial phase of the voting period may not mean much.

Nouns:

> We still see it s a feature to be able to set it to the entire voting period. Won't fix

Spearbit: Our take here is that during initial part of the voting period this trigger is more a bug than a feature. Switching from `Defeated` to `Successful` brings in no information when voting is just started. Acknowledged.

5.4 Gas Optimization

5.4.1 Use custom errors instead of revert strings and remove pre-existing unused custom errors

Severity: Gas Optimization

Context: NounsDAOProxy.sol#L79, NounsDAOExecutorV2.sol#L108, NounsDAOExecutor.sol#L134, NounsDAOLogicV1Fork.sol#L680, NounsTokenFork.sol#L43

Description: String errors are added to the bytecode which make deployment cost more expenseive. It is also difficult to use dynamic information in them. Custom errors are more convenient and gas-efficient.

There are several cases across the codebase where long string errors are still used over custom errors. As an example, in NounsDAOLogicV1Fork.sol#L680, the check reverts with a string:

```
require(msg.sender == admin, 'NounsDAO::_setQuorumVotesBPS: admin only');
```

In this case, the `AdminOnly()` custom error can be used here to save gas. This also occur in other parts of this contract as well as the codebase.

Also, some custom errors were defined but not used. See NounTokenFork.sol#L40, NounTokenFork.sol#L43

Recommendation: It is worth using custom errors over string errors as this will save gas. Review and update the codebase with custom errors where convenient. Also, remove any unused custom error.

Nouns:

> Won't Fix. We do benefit from the current design, e.g. by being able to run the same test code on multiple DAO versions.

Spearbit: Acknowledged.

5.4.2 `escrowedTokensByForkId` can be used to get owner of escrowed tokens

Severity: Gas Optimization

Context: NounsDAOForkEscrow.sol#L58

Description: The state variable `escrowedTokensByForkId` in L58 creates a getter function that can be used to check the owner of escrowed token. This performs the same function as calling `ownerOfEscrowedToken()` and might be considered redundant.

Recommendation: Consider removing the function `ownerOfEscrowedToken()` to reduce code and deployment cost.

Nouns:

> Won't Fix. The deployment gas savings aren't important enough right now.

Spearbit: Acknowledged.

5.4.3 Emit events using locally assigned variables instead of reading from storage to save on SLOAD

Severity: Gas Optimization

Context: NounsDAOV3Admin.sol#L284, NounsDAOLogicV1Fork.sol#L619, NounsDAOExecutorV2.sol#L104, NounsDAOProxy.sol#L85

Description: By emitting local variables over storage variables, when they have the same value, you can save gas on SLOAD.

Some examples include: NounsDAOLogicV1Fork.sol#L619 :

```
-     emit VotingDelaySet(oldVotingDelay, votingDelay);
+     emit VotingDelaySet(oldVotingDelay, newVotingDelay);
```

NounsDAOLogicV1Fork.sol#L635 :

```
-     emit VotingPeriodSet(oldVotingPeriod, votingPeriod);
+     emit VotingPeriodSet(oldVotingPeriod, newVotingPeriod);
```

NounsDAOLogicV1Fork.sol#L653 :

```
-     emit ProposalThresholdBPSSet(oldProposalThresholdBPS, proposalThresholdBPS);
+     emit ProposalThresholdBPSSet(oldProposalThresholdBPS, newProposalThresholdBPS);
```

NounsDAOLogicV1Fork.sol#L670 :

```
-     emit QuorumVotesBPSSet(oldQuorumVotesBPS, quorumVotesBPS);
+ emit QuorumVotesBPSSet(oldQuorumVotesBPS, newQuorumVotesBPS);
```

NounsDAOExecutorV2.sol#L104 :

```
-     emit NewDelay(delay);
+     emit NewDelay(delay_);
```

NounsDAOExecutorV2.sol#L112 :

```
-     emit NewAdmin(admin);
+     emit NewAdmin(msg.sender);
```

NounsDAOExecutorV2.sol#L122 :

```
-     emit NewPendingAdmin(pendingAdmin);
+     emit NewPendingAdmin(pendingAdmin_);
```

NounsDAOV3Admin.sol#L284 :

```
-     emit NewPendingAdmin(oldPendingAdmin, ds.pendingAdmin);
+     emit NewPendingAdmin(oldPendingAdmin, address(0));
```

NounsDAOProxy.sol#L85 :

```
-     emit NewImplementation(oldImplementation, implementation);
+     emit NewImplementation(oldImplementation, implementation_);
```

Recommendation: Review the codebase and consider using the existing local values instead of reading from storage when emitting events. This will save some gas since it will avoid usage of the `SLOAD` opcode.

Nouns:

Fix PR: nounsDAO/nouns-monorepo#736.

Spearbit: Fixed.

5.5 Informational

5.5.1 `joinFork()` violates Checks-Effects-Interactions best practice for reentrancy mitigation

Severity: Informational

Context: NounsDAOV3Fork.sol#L139-L158

Description: `joinFork()` interacts with `forkDAOTreasury` in `sendProRataTreasury()` to send pro rata original DAO treasury for the tokens joining the fork. This interaction with the external `forkDAOTreasury` contract happens before the transfer of the original DAO tokens to the timelock is effected.

While `forkDAOTreasury` is under the control of the fork DAO (outside the trust model of original DAO) and `joinFork()` does not have a reentrancy guard, we do not see a potential/meaningful exploitable reentrancy here.

Recommendation: Implement the Checks-Effects-Interactions best practice by transferring the original tokens to the timelock before sending its proportional treasury from the original DAO to the fork DAO.

Nouns:

Won't Fix. Risk is too low.

Spearbit: Acknowledged.

5.5.2 Rename `MAX_VOTING_PERIOD` and `MAX_VOTING_DELAY` to enhance readability.

Severity: Informational

Context: NounsDAOV3Admin.sol#L115, NounsDAOV3Admin.sol#L121

Description: Given that the state variables `MAX_VOTING_PERIOD` (NounsDAOV3Admin.sol#L115) and `MAX_VOTING_DELAY` (NounsDAOV3Admin.sol#L121) are in blocks, it is more readable if the name has a `_BLOCKS` suffix and is set to 2 weeks / 12 as done with `MAX_OBJECTION_PERIOD_BLOCKS` and `MAX_UPDATABLE_PERIOD_BLOCKS`.

The functions, `_setVotingDelay` (L152) and `_setVotingPeriod` (L167), can be renamed in the same vain by adding `-InBlocks` suffix similar to `_setObjectionPeriodDurationInBlocks` and other functions.

In addition to this, constants should be named with all capital letters with underscores separating words, following the Solidity style guide. For example, `proposalMaxOperations` in NounsDAOV3Proposals.sol#L138 can be renamed.

Recommendation: Implement changes in `MAX_VOTING_PERIOD` and `MAX_VOTING_DELAY` to enhance readability. Rename `proposalMaxOperations` to follow the style guide for constants.

Nouns:

Fixed: Ack, fix PR: PR 728 and commit 6d4388.

Spearbit: Fixed.

5.5.3 External function is used instead of internal equivalent across NounsDAOV3Proposals logic

Severity: Informational

Context: NounsDAOV3Proposals.sol#L389

Description: Public view `state(ds, proposalId)` is used instead of the fully equivalent internal `stateInternal(ds, proposalId)` in the several occurrences of NounsDAOV3Proposals logic.

For example, in the `NounsDAOV3Proposals:updateProposalBySigs` the `state(ds, proposalId)` is used instead of the `stateInternal` function:

```
        if (state(ds, proposalId) != NounsDAOStorageV3.ProposalState.Updatable) revert
        ↪  CanOnlyEditUpdatableProposals();
```

Recommendation: It is recommended to use `stateInternal(ds, proposalId)` instead of the public `state(ds, proposalId)` function.

Nouns:

Fix PR: nounsDAO/nouns-monorepo#729.

Spearbit: Fix looks good.

5.5.4 Proposals created through `proposeBySigs()` can not be executed on `TimelockV1`

Severity: Informational

Context: NounsDAOV3Proposals.sol#L204

Description: Currently, proposals created through the `proposeBySigs()` function can not be executed on `TimelockV1`. This could potentially limit the flexibility of creating different types of proposals.

It may be advantageous to have a parameter added to the `proposeBySigs()` function, allowing the proposer to decide whether the proposal should be executed on `TimelockV1` or not. There's a design decision to be made regarding whether this value should be incorporated as part of the signers' signature, or simply left up to the proposer to determine if execution should happen on the `TimelockV1` or not.

Recommendation: Consider modifying the function by adding a parameter for execution location decision. This could be part of the signer's signature or be at the proposer's discretion.

Nouns: Acknowledged, won't fix.

Spearbit: Acknowledged.

5.5.5 `escrowToFork()` can be frontrun to prevent users from joining the fork during the escrow period

Severity: Informational

Context: NounsDAOV3Fork.sol#L72-L86 NounsDAOV3Fork.sol#L109-L130

Description: `escrowToFork()` could be frontrun by another user and make it revert by either:

1. Frontrunning with another `escrowToFork()` that reaches the fork threshold + `executeFork()`.
2. If the fork threshold was already reached, frontrunning with `executeFork()`.

This forces the escrowing user to join the fork only during the forking period.

Recommendation: Consider a delay period between the escrow and forking periods so that `executeFork()` is a time-based trigger instead of relying only on the threshold.

Nouns:

Thank you, we won't fix this one.

Spearbit: Acknowledged.

5.5.6 Fork spec says Nouns are escrowed during the fork active period

Severity: Informational

Context: Fork-Spec

Description: Fork-Spec says: "*During the forking period additional forking Nouns are also sent to the escrow contract; the motivation is to have a clean separation between fork-related Nouns and Nouns owned by the DAO for other reasons.*" However, the implementation sends such Nouns to the original DAO treasury.

Recommendation: Update the spec to reflect the current implementation.

Nouns: Spec is updated.

Spearbit: This has been updated in the latest spec commit.

5.5.7 Known issues from previous versions/audit

Severity: Informational

Context: Audit-Report Note

Description: Below are some of the known issues from previous versions as reported by the protocol team, documented in the audit report and is being recorded here verbatim for reporting purposes.

Note: All issues reported earlier and not fixed in current version(s) of audit scope are assumed to be acknowledged without fixing.

> NounsToken delegateBySigs allows delegating to address zero We've fixed this issue in the fork token contract, but cannot fix it in the original NounsToken because the contract isn't upgradeable.
>
> Voting gas refund can be abused We're aware of different ways of abusing this function:
>
> A token holder could delegate their Nouns to a contract and vote on multiple proposals in a loop, such that the tx gas overhead is amortized across all votes, while the refund function assumes each vote has the full overhead to bear; this could result in token holders profiting from gas refunds. A token holder could grief the DAO by voting with very long reason strings, in order to drain the refund balance faster. We find these issues low risk and unlikely given the small size of the community, and the low ETH balance the governor contract has to spend on refunds. Should we see such abusive activity, we might reconsider this feature.
>
> Nouns transfers will stop working when block number hits uint32 max value We're aware of this issue. It means the Nouns token will stop functioning a long long long time from now :)
>
> AuctionHouse has an open gas griefing vector Bidder Alice can bid from a smart contract that returns a large byte array when receiving ETH. Then if Bob outbids Alice, in his bid tx AuctionHouse refunds Alice, and the large return value causes a gas cost spike for Bob. See more details here.
>
> We're planning to fix this in the next AuctionHouse version, its launch date is unknown at this point.
>
> Using error strings instea of custom errors In all new code we're using custom errors. In code that's forked from previous versions we optimized for the smallest diff possible, and so leaving error strings untouched.

Recommendation: Nouns: Acknowledged.

Spearbit: Acknowledged

5.5.8 When a minority forks, the majority can follow

Severity: Informational

Context: When a minority forks, the majority can follow

Description: This is a known issue as documented by the protocol team and is being recorded here verbatim for reporting purposes.

> For example, a malicious majority can vote For a proposal to drain the treasury, forcing others to fork; the majority can then join the fork with many of their tokens, benefiting from the passing proposal on the original DAO, while continuing to attack the minority in their new fork DAO, forcing them to quit the fork DAO.
>
> This is a well known flaw of the current fork design, something we've chosen to go live with for the sake of shipping something the DAO has asked for urgently.

Recommendation: Consider a redesign which mitigates this issue in a future iteration.

Nouns: Acknowledged.

Spearbit: Acknowledged.

5.5.9 The original DAO can temporarily brick a fork DAO's token minting

Severity: Informational

Context: The-original-DAO-can-temporarily-brick-a-fork DAO's-token-minting

Description: This is a known issue as documented by the protocol team and is being recorded here verbatim for reporting purposes.

> We've decided in this version to deploy fork DAOs such that fork tokens reuse the same descriptor contract as the original DAO's token descriptor. Our motivations are minimizing lines of code and the gas cost of deploying a fork.
>
> This design poses a risk on fork tokens: the original DAO can update the descriptor to use a new art contract that always reverts, which would then lead to fork token's mint function always reverting.
>
> The solution would be for the fork DAO to execute a proposal that deploys and sets a new descriptor to its token, which would use a valid art contract, allowing minting to resume.
>
> The fork DAO is guaranteed to be able to propose and execute such a proposal, because the function where Nouners claim their fork tokens does not use the descriptor, and so is not vulnerable to this attack.

Recommendation: Ensure that the forking UI/UX provides guidance on this aspect.

Nouns: Acknowledged.

Spearbit: Acknowledged.

5.5.10 Unused events, missing events and unindexed event parameters in contracts

Severity: Informational

Context: INounsTokenFork.sol#L29, NounsDAOV3Admin.sol#L509-514, NounsDAOEventsFork.sol#L43

Description: Some contracts have missing or unused events, as well as event parameters that are unindexed. As an examples:

1. Unused events: INounsTokenFork.sol#L29: `event NoundersDAOUpdated(address noundersDAO);`
2. Missing events: NounsDAOV3Admin.sol#L509-514: Missing event like in `_setForkEscrow`
3. Unindexed parameters: NounsDAOEventsFork.sol: Many parameters can be indexed here

Recommendation: Review the codebase, remove unused events, add missing events and index event parameters where necessary

Nouns:

> Fix PR: nounsDAO/nouns-monorepo#730.

Spearbit: Fixed.

5.5.11 Prefer using `__Ownable_init` instead of `_transferOwnership` to initialize upgradable contracts

Severity: Informational

Context: NounsAuctionHouseFork.sol#L87, NounsTokenFork.sol#L129

Description: The upgradable `NounsAuctionHouseFork` and the `NounsTokenFork` contracts inherit the `OwnableUpgradeable` contract. However, inside the `initialize` function the ownership transfer is performed by calling the internal `_transferOwnership` function instead of calling the `__Ownable_init`. This deviates from the standard approach of initializing upgradable contracts, and it can lead to issues if the `OwnableUpgradeable` contract changes its initialization mechanism.

Recommendation: It is recommended to use the `__Ownable_init` function instead of `_transferOwnership` to initialize the mentioned contracts.

Nouns:

> Won't Fix.
>
> We're still using an older OZ version without the ability to set a custom owner in the init function. Upgrading our OZ version is out of scope in this release.

Spearbit: Acknowledged.

5.5.12 Consider emitting the address of the timelock in the `ProposalQueued` event

Severity: Informational

Context: NounsDAOV3Proposals.sol#L448

Description: The `queue` function currently emits the `ProposalQueued` event to provide relevant information about the proposal, including the `proposalId` and the `eta`. However, it doesn't emit the `timelock` variable, which represents the address of the timelock responsible for executing the proposal. This could lead to confusion among users regarding the intended `timelock` for proposal execution.

Recommendation: To address this issue, it is recommended to include the address of the `timelock` in the `ProposalQueued` event as well. This improvement will enhance the clarity and usability of the emitted event.

Nouns:

> We won't change the `ProposalQueued` event at this to maintain backwards compatibility with Governor Bravo clients. We'll use the event `ProposalCreatedOnTimelockV1` for this purpose during creation time.

Spearbit: Acknowledged.

5.5.13 Use `IERC20Upgradeable/IERC721Upgradeable` for consistency with other contracts

Severity: Informational

Context: NounsAuctionHouseFork.sol#L35, NounsDAOLogicV1Fork.sol#L102, NounsTokenFork.sol#L25

Description: Most contracts/libraries imported and used are the upgradeable variant e.g. `OwnableUpgradeable`. `IERC20` and `IERC721` are used which is inconsistent with the other contracts/libraries. Since the project is deployed with upgradeability featured, it is more preferable to use the Upgradeable variant of OpenZeppelin Contracts.

Recommendation: Use the `IERC20Upgradeable` or `IERC721Upgradeable` version where applicable in the codebase to be consistent with other upgradeable contracts/libraries.

Nouns:

> Won't Fix.

Spearbit: Acknowledged.

5.5.14 Specification says "Pending" state instead of "Updatable"

Severity: Informational

Context: nouns-dao-v3-spec

Description: The V3 spec says the following for "Proposal editing": "*The proposer account of a proposal in the PENDING state can call an updateProposal function, providing the new complete set of transactions to execute, as well as a complete new version of the proposal description text.*"

This is incorrect because editing can only happen in the "Updatable" state which is just before the "Pending" state.

Recommendation: Fix the specification document to say: "*The proposer account of a proposal in the UPDATABLE state can call an updateProposal function...*". Consider adding more details in the specification corresponding to what's been implemented.

Nouns: Fixed.

Spearbit: Verified that the spec has been updated as per recommendation.

5.5.15 Typos, comments and descriptions need to be updated

Severity: Informational

Context: NounsDAOLogicV1Fork.sol#L48, NounsDAOLogicV1Fork.sol#L80, NounsDAOV3Votes.sol#L267, NounsDAOExecutorProxy.sol#L24, NounsDAOV3Admin.sol#L130, NounsTokenFork.sol#L66, NounsDAOV3DynamicQuorum.sol#L124, NounsDAOStorageV1Fork.sol#L33, NounsDAOLogicV3.sol#L902

Description: The contract source code contains several typographical errors and misaligned comments/descriptions.

Typos:

1. NounsDAOLogicV1Fork.sol#L48: `adjutedTotalSupply` should be `adjustedTotalSupply`
2. NounsDAOLogicV1Fork.sol#L80: `veteor` shoud be `vetoer`
3. NounsDAOV3Votes.sol#L267: `objetion` should be `objection`
4. NounsDAOExecutorProxy.sol#L24: `imlemenation` should be `implementation`

Comment Discrepancies:

1. NounsDAOV3Admin.sol#L130: Should say `// 6,000 basis points or 60%` and not `4,000`
2. NounsDAOV3Votes.sol#L219: change string `'NounsDAO::castVoteInternal: voter already voted'` to `'NounsDAO::castVoteDuringVotingPeriodInternal: voter already voted'`

3. NounsDAOExecutorV2.sol#L209: change string `'NounsDAOExecutor::executeTransaction: Call must come from admin.` to `'NounsDAOExecutor::sendETH: Call must come from admin.`
4. NounsDAOExecutorV2.sol#L221: change string `NounsDAOExecutor::executeTransaction: Call must come from admin.` to `NounsDAOExecutor::sendERC20: Call must come from admin.`
5. NounsDAOV3DynamicQuorum.sol#L124: Should be adjusted total supply
6. NounsDAOV3DynamicQuorum.sol#L135: Should be adjusted total supply
7. NounsDAOLogicV3.sol#L902: Adjusted supply is used for `minQuorumVotes()`
8. NounsDAOLogicV3.sol#L909: Adjusted supply is used for `maxQuorumVotes()`
9. NounsDAOStorageV1Fork.sol#L33: `proposalThresholdBPS` is required to be exceeded, say when it is zero, one noun is needed to propose
10. NounsTokenFork.sol#L66: Typo, to be `after which`

Recommendation: It is recommended to fix the typographical errors and update the comments to align with the implementation. You might consider utilizing a tool such as codespell to aid in the identification of misspelled words within the source code.

Nouns:

> Fix PR: nounsDAO/nouns-monorepo#731.

Spearbit: Fixed.

5.5.16 Contracts are not using the `_disableInitializers` function

Severity: Informational

Context: NounsDAOExecutorV2.sol#L46, NounsAuctionHouseFork.sol#L84, NounsTokenFork.sol#L127

Description: Several Nouns-Dao contracts utilize the Initializable module provided by OpenZeppelin. To ensure that an implementation contract is not left uninitialized, it is recommended in OpenZeppelin's documentation to include the `_disableInitializers` function in the constructor. The `_disableInitializers` function automatically locks the contracts upon deployment.

According to the `OpenZeppelin` documentation:

> Do not leave an implementation contract uninitialized. An uninitialized implementation contract can be taken over by an attacker, which may impact the proxy. To prevent the implementation contract from being used, you should invoke the `_disableInitializers` function in the constructor to automatically lock it when it is deployed:

Recommendation: It is recommended to add the `_disableInitializers` function in the constructor of the upgradeable contracts.

Nouns:

> Won't Fix.
>
> We're still using an older OZ version without this feature. Upgrading our OZ version is out of scope in this release.

Spearbit: Acknowledged.

5.5.17 Missing or incomplete Natspec documentation

Severity: Informational

Context: NounsDAOV3Fork.sol#L203, NounsDAOV3Votes.sol#L295, NounsDAOV3Admin.sol, NounsDAOV3Proposals.sol, NounsDAOExecutor.sol, NounsDAOExecutorV2.sol, NounsTokenFork.sol, NounsDAOV3Admin.sol, NounsDAOV3Proposals.sol, NounsDAOLogicV3.sol

Description: There are several instances throughout the codebase where NatSpec is either missing or incomplete.

1. Missing Natspec (Some functions in this case are missing Natspec comment):
   - NounsDAOV3Fork.sol#L203
   - NounsDAOV3Votes.sol#L295
   - NounsDAOV3Admin.sol
   - NounsDAOV3Proposals.sol
   - NounsDAOExecutor.sol
   - NounsDAOExecutorV2.sol
2. Incomplete Natspec (Some functions are missing `@param` tag):
   - NounsTokenFork.sol
   - NounsDAOV3Admin.sol
   - NounsDAOV3Proposals.sol
   - NounsDAOLogicV3.sol

Recommendation: Consider reviewing the codebase for missing or incomplete NatSpec comments and parameter tags. Add them where appropriate.

Nouns:

> We made some fixes for things we found. Since not all cases were listed we were not able to fix everything. Here are a few:

- PR 737.
- PR 731.

Spearbit: Acknowledged.

5.5.18 Function ordering does not follow the Solidity style guide

Severity: Informational

Context: ERC721CheckpointableUpgradeable.sol#L50, NounsDAOExecutorV2.sol#L46

Description: The recommended order of functions in Solidity, as outlined in the Solidity style guide, is as follows: `constructor()`, `receive()`, `fallback()`, `external`, `public`, `internal` and `private`. However, this ordering isn't enforced in the across the Nouns-Dao codebase.

Recommendation: It is recommended to follow the recommended order of functions in Solidity, as outlined in the Solidity style guide.

Nouns: We won't fix this one.

Spearbit: Acknowledged.

5.5.19 Use a more recent Solidity version

Severity: Informational

Context: ERC721CheckpointableUpgradeable.sol#L46, NounsDAOExecutorV2.sol#L40, NounsDAOLogicV3.sol#L56, NounsDAOV3Votes.sol#L18

Description: The compiler version used `0.8.6` is quite old (current version is `0.8.20`). This version was released almost two years ago and there have been five applicable bug fixes to this version since then. While it seems that those bugs don't apply to the Nouns-Dao codebase, it is advised to update the compiler to a newer version.

Recommendation: It is recommended to upgrade the codebase to a more recent compiler version.

Nouns: We upgraded the solidity version to `0.8.19` in the following PR 724.

Spearbit: Fix looks good. The project now uses solidity version `0.8.19`.

5.5.20 State modifications after external interactions make NounsDAOForkEscrow's `returnTokensToOwner` prone to reentrancy attacks

Severity: Informational

Context: NounsDAOForkEscrow.sol#L110-L125

Description: External NFT call happens before `numTokensInEscrow` update in `returnTokensToOwner()`.

This looks safe (NFT is fixed to be noun contract and `transferFrom()` is used instead of the safe version, and also `numTokensInEscrow = 0` in closeEscrow() acts as a control for `numTokensInEscrow -= tokenIds.length` logic), but in general this type of execution flow structuring could allow for direct stealing via reentrancy.

I.e. in a presence of callback (e.g. arbitrary NFT instead of noun contract or `safeTransferFrom` instead of `transferFrom`) and without `numTokensInEscrow = 0` conflicting with `numTokensInEscrow -= tokenIds.length`, as an abstract example, an attacker would add the last needed noun for forking, then call `withdrawFromForkEscrow()` and then, being in `returnTokensToOwner()`, call `executeFork()` from the callback hook, successfully performing the fork, while already withdrawn the NFT that belongs to DAO.

- NounsDAOForkEscrow.sol#L110-L125)

```
    function returnTokensToOwner(address owner, uint256[] calldata tokenIds) external onlyDAO {
        for (uint256 i = 0; i < tokenIds.length; i++) {
            if (currentOwnerOf(tokenIds[i]) != owner) revert NotOwner();

>>          nounsToken.transferFrom(address(this), owner, tokenIds[i]);
            escrowedTokensByForkId[forkId][tokenIds[i]] = address(0);
        }

        numTokensInEscrow -= tokenIds.length;
    }
```

- NounsDAOV3Fork.sol#L109-L130

```
    function executeFork(NounsDAOStorageV3.StorageV3 storage ds)
        external
        returns (address forkTreasury, address forkToken)
    {
        if (isForkPeriodActive(ds)) revert ForkPeriodActive();
        INounsDAOForkEscrow forkEscrow = ds.forkEscrow;

>>      uint256 tokensInEscrow = forkEscrow.numTokensInEscrow();
        if (tokensInEscrow <= forkThreshold(ds)) revert ForkThresholdNotMet();

        uint256 forkEndTimestamp = block.timestamp + ds.forkPeriod;

        (forkTreasury, forkToken) = ds.forkDAODeployer.deployForkDAO(forkEndTimestamp, forkEscrow);
        sendProRataTreasury(ds, forkTreasury, tokensInEscrow, adjustedTotalSupply(ds));
        uint32 forkId = forkEscrow.closeEscrow();

        ds.forkDAOTreasury = forkTreasury;
        ds.forkDAOToken = forkToken;
        ds.forkEndTimestamp = forkEndTimestamp;

        emit ExecuteFork(forkId, forkTreasury, forkToken, forkEndTimestamp, tokensInEscrow);
    }
```

Direct stealing as a result of state manipulations is possible conditional on an ability to enter a callback. Given the absense of the latter at the moment, but critical impact of the former, considering this as best practice recommendation and setting the severity to be informational.

Recommendation: Consider placing the external call after updating the state (do checks, effects, then interactions), i.e. performing 1 increment per time, for example:

- NounsDAOForkEscrow.sol#L116-L125

```
    function returnTokensToOwner(address owner, uint256[] calldata tokenIds) external onlyDAO {
        for (uint256 i = 0; i < tokenIds.length; i++) {
            if (currentOwnerOf(tokenIds[i]) != owner) revert NotOwner();

-           nounsToken.transferFrom(address(this), owner, tokenIds[i]);
            escrowedTokensByForkId[forkId][tokenIds[i]] = address(0);
+           numTokensInEscrow--;

+           nounsToken.transferFrom(address(this), owner, tokenIds[i]);
        }

-       numTokensInEscrow -= tokenIds.length;
    }
```

It does add storage manipulations and increases gas cost, but it can be reasonable for this somewhat rarely called function that performs only few iterations in the majority of cases.

Nouns:

> Won't Fix. Given the assumption of using NounsToken without external calls, the risk to Nouns DAO is zero.

Spearbit: Acknowledged.

5.5.21 No need to use an assembly block to get the chain ID

Severity: Informational

Context: ERC721CheckpointableUpgradeable.sol#L294-L300

Description: Currently the `getChainId()` uses an assembly block to get the current chain ID when constructing the domain separator. This is not needed since there is a global variable for this already.

Recommendation: You could consider removing the internal `getChainId()` method and use the global variable `block.chainid` instead.

Nouns: Fixed in this commit: 1fc830

Spearbit: Verified.

5.5.22 Naming convention for interfaces is not always always followed

Severity: Informational

Context: NounsTokenForkLike.sol#L5

Description: The `NounsTokenForkLike` interface does not follow the standard naming convention for Solidity interfaces, which begins with an `I` prefix. This inconsistency can make it harder for developers to understand the purpose and usage of the contract.

Recommendation: It is recommended to prefix the `NounsTokenForkLike` interface with `I`:

```
-interface NounsTokenForkLike {
+interface INounsTokenForkLike {
    function getPriorVotes(address account, uint256 blockNumber) external view returns (uint96);

    function totalSupply() external view returns (uint256);

    . . .
```

Nouns: Fixed in this commit ede514

Spearbit: Verified. The interface is now prefixed with `I`.