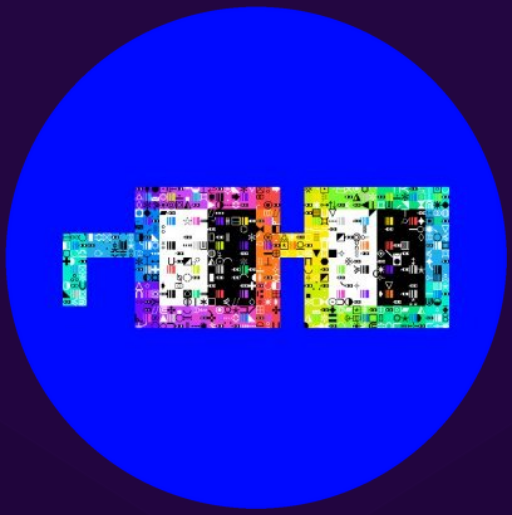
SHERLOCK

SHERLOCK SECURITY REVIEW FOR



Prepared for: Nouns Builder
Prepared by: Sherlock
Lead Security Expert: 0x52
Dates Audited: November 22 - December 1, 2023
Prepared on: January 3, 2024

Introduction

Nouns Builder is a tool that allows any DAO to form and govern completely onchain, in the format of Nouns DAO. Nouns are an experimental attempt to improve the formation of onchain avatar communities.

Scope

Repository: ourzora/nouns-protocol

Branch: v2-audit-min

Commit: e81cfce40e09b8abd9222443373ac747598bac4b

For the detailed scope, see the contest details.

Findings

Each issue has an assigned severity:

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

Issues found

Medium	High
3	2

Issues not fixed or acknowledged

Medium	High
0	0

Security experts who found valid issues

SilentDefendersOfDeFi
0x52
jerseyjoewalcott
nirohgo
unforgiven
ggg_ttt_hhh
KupiaSec
dany.armstrong90
xAriextz
HHK
pontifex
coffiasd
gqrp
Kow
Brenzee
cu5t0mPe0
Bauer
giraffe
popeye
0xMosh
Inspex
Tricko
kutugu
Krace
0xbepresent
Nyx
dimulski
Aamirusmani1552
0xmystery
Juntao
almurhasan
rvierdiiev
SovaSlava
deepkin
qpzm
KingNFT
0xpep7
ydlee
circlelooper
bin2chen
Falconhoof
cawfree
Jiamin
chaduke
0xcrunch
Ch_301
saian
ast3ros
0xReiAyanami
ge6a
zraxx
whoismxuse

Issue H-1: when reservedUntilTokenId > 100 first funder loss 1% NFT

Source: https://github.com/sherlock-audit/2023-09-nounsbuilder-judging/issues/42

Found by

0x52, 0xReiAyanami, 0xbepresent, 0xcrunch, 0xmystery, 0xpep7, Aamirusmani1552, Ch_301, Falconhoof, HHK, Jiamin, Juntao, KingNFT, Kow, Krace, KupiaSec, Nyx, SilentDefendersOfDeFi, SovaSlava, almurhasan, ast3ros, bin2chen, cawfree, chaduke, circlelooper, coffiasd, dany.armstrong90, deepkin, dimulski, ge6a, ggg_ttt_hhh, giraffe, gqrp, pontifex, qpzm, rvierdiiev, saian, unforgiven, whoismxuse, xAriextz, ydlee, zraxx

Summary

The incorrect use of `baseTokenId = reservedUntilTokenId` may result in the first `tokenRecipient[]` being invalid, thus preventing the founder from obtaining this portion of the NFT.

Vulnerability Detail

The current protocol adds a parameter `reservedUntilTokenId` for reserving `Token`. This parameter will be used as the starting `baseTokenId` during initialization.

```
    function _addFounders(IManager.FounderParams[] calldata _founders, uint256
↪   reservedUntilTokenId) internal {
...

                // Used to store the base token id the founder will recieve
@>              uint256 baseTokenId = reservedUntilTokenId;

                // For each token to vest:
                for (uint256 j; j < founderPct; ++j) {
                    // Get the available token id
                    baseTokenId = _getNextTokenId(baseTokenId);

                    // Store the founder as the recipient
                    tokenRecipient[baseTokenId] = newFounder;

                    emit MintScheduled(baseTokenId, founderId, newFounder);

                    // Update the base token id
                    baseTokenId = (baseTokenId + schedule) % 100;
```

```
                }
            }
..

    function _getNextTokenId(uint256 _tokenId) internal view returns (uint256) {
        unchecked {
@>          while (tokenRecipient[_tokenId].wallet != address(0)) {
                _tokenId = (++_tokenId) % 100;
            }

            return _tokenId;
        }
    }
```

Because `baseTokenId = reservedUntilTokenId` is used, if `reservedUntilTokenId>100`, for example, reservedUntilTokenId=200, the first `_getNextTokenId(200)` will return `baseTokenId=200` , `tokenRecipient[200]=newFounder`.

Example: reservedUntilTokenId = 200 founder[0].founderPct = 10

In this way, the `tokenRecipient[]` of `founder` will become tokenRecipient[200].wallet = founder (first will call _getNextTokenId(200) return 200) tokenRecipient[10].wallet = founder (second will call _getNextTokenId((200 + 10) %100 = 10)) tokenRecipient[20].wallet = founder ... tokenRecipient[90].wallet = founder

However, this `tokenRecipient[200]` will never be used, because in `_isForFounder()`, it will be modulo, so only `baseTokenId < 100` is valid. In this way, the first founder can actually only 9% of NFT.

```
    function _isForFounder(uint256 _tokenId) private returns (bool) {
        // Get the base token id
@>      uint256 baseTokenId = _tokenId % 100;

        // If there is no scheduled recipient:
        if (tokenRecipient[baseTokenId].wallet == address(0)) {
            return false;

            // Else if the founder is still vesting:
        } else if (block.timestamp < tokenRecipient[baseTokenId].vestExpiry) {
            // Mint the token to the founder
@>          _mint(tokenRecipient[baseTokenId].wallet, _tokenId);

            return true;

            // Else the founder has finished vesting:
```

```
        } else {
            // Remove them from future lookups
            delete tokenRecipient[baseTokenId];

            return false;
        }
    }
```

POC

The following test demonstrates that `tokenRecipient[200]` is for founder.

1. need change tokenRecipient to public , so can assertEq

```
contract TokenStorageV1 is TokenTypesV1 {
    /// @notice The token settings
    Settings internal settings;

    /// @notice The vesting details of a founder
    /// @dev Founder id => Founder
    mapping(uint256 => Founder) internal founder;

    /// @notice The recipient of a token
    /// @dev ERC-721 token id => Founder
-   mapping(uint256 => Founder) internal tokenRecipient;
+   mapping(uint256 => Founder) public tokenRecipient;
}
```

2. add to `token.t.sol`

```
function test_lossFirst(address _minter, uint256 _reservedUntilTokenId, uint256
↪   _tokenId) public {
    deployAltMock(200);
    (address wallet ,,)= token.tokenRecipient(200);
    assertEq(wallet,founder);
}
```

```
$ forge test -vvv --match-test test_lossFirst

Running 1 test for test/Token.t.sol:TokenTest
[PASS] test_lossFirst(address,uint256,uint256) (runs: 256, : 3221578, ~: 3221578)
Test result: ok. 1 passed; 0 failed; 0 skipped; finished in 355.45ms
Ran 1 test suites: 1 tests passed, 0 failed, 0 skipped (1 total tests)
```

Impact

when reservedUntilTokenId > 100 first funder loss 1% NFT

Code Snippet

https://github.com/sherlock-audit/2023-09-nounsbuilder/blob/main/nouns-protocol/src/token/Token.sol#L161

Tool used

Manual Review

Recommendation

1. A better is that the baseTokenId always starts from 0.

```
    function _addFounders(IManager.FounderParams[] calldata _founders, uint256
↪   reservedUntilTokenId) internal {
...

                // Used to store the base token id the founder will recieve
-               uint256 baseTokenId = reservedUntilTokenId;
+               uint256 baseTokenId =0;
```

or

2. use `uint256 baseTokenId = reservedUntilTokenId  % 100;`

```
    function _addFounders(IManager.FounderParams[] calldata _founders, uint256
↪   reservedUntilTokenId) internal {
...

                // Used to store the base token id the founder will recieve
-               uint256 baseTokenId = reservedUntilTokenId;
+               uint256 baseTokenId = reservedUntilTokenId  % 100;
```

Discussion

neokry

This is valid and is the core issue behind #247 as well. baseTokenId should start at 0 in `addFounders`

nevillehuang

I initially separated the 4 findings below, but I agree, #177, #247 and #67 are only possible because of the following lines of code here, wherein `_addFounder()`, `baseTokenId` is incorrectly initialized to `reservedUntilTokenId` in `addFounders()`, which is the root cause of the issue, and once fixed, all the issues will be fixed too. There are 4 impacts mentioned by watsons.

```
uint256 baseTokenId = reservedUntilTokenId;
```

1. Previous founders that are meant to be deleted are retained causing them to continue receiving minted NFTs --> High severity, since it is a definite loss of funds
2. #247: Any `reserveTokenId` greater than 100 will cause a 1% loss of NFT for founder --> High severity, since it is a definite loss of funds for founder as long as `reservedUntilTokenId` is set greater than 100, which is not unlikely
3. #177: This is essentially only an issue as `baseTokenId` is incorrectly set as `reservedUntilTokenId` but will cause a definite loss of founders NFT if performed, so keeping as duplicate
4. #67: This is closely related to the above finding (177), where a new update to `reservedUntilTokenId` via `setReservedUntilTokenId` can cause over/underallocation NFTs so keeping as duplicate

However, in the context of the audit period, I could also see why watsons separated these issues, so happy to hear from watsons during escalation period revolving deduplication of these issues.

neokry

Fixed here: https://github.com/ourzora/nouns-protocol/pull/122

nevillehuang

Hi @neokry would be helpful if you could highlight to watsons here why you think the following primary issues should be duplicated under this issue:

#67 #177 #247

From my understanding it stems from the `_addFounders()` function used in both the `initialize()` and `updateFounders()` function, in particular the following line here,

```
uint256 baseTokenId = reservedUntilTokenId;
```

But it would be extremely helpful if you could provide a more detailed explanation in each finding, and show how the fix to #42 also fixes the rest of the findings.

To all watsons, this is my initial deduplication here, feel free to also provide me the flow state of the functions to prove that they do not have the same root cause.

nevillehuang

Hi watsons, The core of issue #42 is that `baseTokenId` should not start with `reservedUntilTokenId` within `addFounders()`

#67 and its duplicates I believe this issue and its duplicates are invalid as there is a misunderstanding of how founders token amount are assigned based on this comment here

Both #177 and #247 and its duplicates This issue hinges on the same root cause that `baseTokenId` is initialized as `reservedUntilTokenId` . However, the key difference here is that `updateFounders()` is also affected, which is a completely different function. However, I still think that this should be duplicated with #42, based on sherlock duplication rules, more specifically, see point 1.1 and 2. The only point where they cannot be considered duplicates is when the fixes are different.

Unless a watson can prove to me that the fix implemented here by the sponsor is insufficient, I am inclined to keep all of them as duplicates except the above mentioned #67 and its duplicates.

IAm0x52

Fix looks good. BaseTokenId now always starts at 0.

Issue H-2: Adversary can permanently brick auctions due to precision error in Auction#_computeTotalRewards

Source: https://github.com/sherlock-audit/2023-09-nounsbuilder-judging/issues/251

Found by

0x52, Bauer, Brenzee, HHK, Kow, KupiaSec, SilentDefendersOfDeFi, coffiasd, cu5t0mPe0, dany.armstrong90, ggg_ttt_hhh, gqrp, pontifex, unforgiven, xAriextz

Summary

When batch depositing to ProtocolRewards, the msg.value is expected to match the sum of the amounts array EXACTLY. The issue is that due to precision loss in Auction#_computeTotalRewards this call can be engineered to always revert which completely bricks the auction process.

Vulnerability Detail

ProtocolRewards.sol#L55-L65

```
for (uint256 i; i < numRecipients; ) {
    expectedTotalValue += amounts[i];

    unchecked {
        ++i;
    }
}

if (msg.value != expectedTotalValue) {
    revert INVALID_DEPOSIT();
}
```

When making a batch deposit the above method is called. As seen, the call with revert if the sum of amounts does not EXACTLY equal the msg.value.

Auction.sol#L474-L507

```
uint256 totalBPS = _founderRewardBps + referralRewardsBPS + builderRewardsBPS;

...

// Calulate total rewards
split.totalRewards = (_finalBidAmount * totalBPS) / BPS_PER_100_PERCENT;
```

```
...

// Initialize arrays
split.recipients = new address[](arraySize);
split.amounts = new uint256[](arraySize);
split.reasons = new bytes4[](arraySize);

// Set builder reward
split.recipients[0] = builderRecipient;
split.amounts[0] = (_finalBidAmount * builderRewardsBPS) / BPS_PER_100_PERCENT;

// Set referral reward
split.recipients[1] = _currentBidRefferal != address(0) ? _currentBidRefferal :
↪  builderRecipient;
split.amounts[1] = (_finalBidAmount * referralRewardsBPS) / BPS_PER_100_PERCENT;

// Set founder reward if enabled
if (hasFounderReward) {
    split.recipients[2] = founderReward.recipient;
    split.amounts[2] = (_finalBidAmount * _founderRewardBps) /
    ↪  BPS_PER_100_PERCENT;
}
```

The sum of the percentages are used to determine the totalRewards. Meanwhile, the amounts are determined using the broken out percentages of each. This leads to unequal precision loss, which can cause totalRewards to be off by a single wei which cause the batch deposit to revert and the auction to be bricked. Take the following example:

Assume a referral reward of 5% (500) and a builder reward of 5% (500) for a total of 10% (1000). To brick the contract the adversary can engineer their bid with specific final digits. In this example, take a bid ending in 19.

```
split.totalRewards = (19 * 1,000) / 100,000 = 190,000 / 100,000 = 1

split.amounts[0] = (19 * 500) / 100,000 = 95,000 / 100,000 = 0
split.amounts[1] = (19 * 500) / 100,000 = 95,000 / 100,000 = 0
```

Here we can see that the sum of amounts is not equal to totalRewards and the batch deposit will revert.

Auction.sol#L270-L273

```
if (split.totalRewards != 0) {
    // Deposit rewards
```

```
    rewardsManager.depositBatch{ value: split.totalRewards }(split.recipients,
    ↪  split.amounts, split.reasons, "");
}
```

The depositBatch call is placed in the very important _settleAuction function. This results in auctions that are permanently broken and can never be settled.

Impact

Auctions are completely bricked

Code Snippet

Auction.sol#L244-L289

Tool used

Manual Review

Recommendation

Instead of setting totalRewards with the sum of the percentages, increment it by each fee calculated. This way they will always match no matter what.

Discussion

nevillehuang

Contrary to #103 which only affects bidding, this causes a complete DoS of settlement of auctions, forcing the DAO to possibly have to redeploy to resolve the issue and continue auctions, so I believe high severity is fair.

neokry

Fixed here: https://github.com/ourzora/nouns-protocol/pull/123

IAm0x52

Fix looks good. split.totalRewards is now calculated incrementally so it will guaranteed match.

Issue M-1: Attacker can force pause the Auction contract.

Source: https://github.com/sherlock-audit/2023-09-nounsbuilder-judging/issues/243

Found by

0xMosh, Inspex, SilentDefendersOfDeFi, Tricko, giraffe, kutugu, popeye, unforgiven

Summary

In certain situations (e.g founders have ownership percentage greater than 51) an attacker can potentially exploit the `try catch` within the `Auction._CreateAuction()` function to arbitrarily pause the auction contract.

Vulnerability Detail

Consider the code from `Auction._CreateAuction()` function, which is called by `Auction.settleCurrentAndCreateNewAuction()`. It first tries to mint a new token for the auction, and if the minting fails the `catch` branch will be triggered, pausing the auction.

```
function _createAuction() private returns (bool) {
    // Get the next token available for bidding
    try token.mint() returns (uint256 tokenId) {
        // Store the token id
        auction.tokenId = tokenId;

        // Cache the current timestamp
        uint256 startTime = block.timestamp;

        // Used to store the auction end time
        uint256 endTime;

        // Cannot realistically overflow
        unchecked {
            // Compute the auction end time
            endTime = startTime + settings.duration;
        }

        // Store the auction start and end time
        auction.startTime = uint40(startTime);
        auction.endTime = uint40(endTime);
```

```
        // Reset data from the previous auction
        auction.highestBid = 0;
        auction.highestBidder = address(0);
        auction.settled = false;

        // Reset referral from the previous auction
        currentBidReferral = address(0);

        emit AuctionCreated(tokenId, startTime, endTime);
        return true;
    } catch {
        // Pause the contract if token minting failed
        _pause();
        return false;
    }
}
```

Due to the internal logic of the mint function, if there are founders with high ownership percentages, many tokens can be minted to them during calls to `mint`as part of the vesting mechanism. As a consequence of this under some circumstances calls to `mint` can consume huge amounts of gas.

Currently on Ethereum and EVM-compatible chains, calls can consume at most 63/64 of the parent's call gas (See EIP-150). An attacker can exploit this circumstances of high gas cost to restrict the parent gas call limit, making `token.mint()` fail and still leaving enough gas left (1/64) for the `_pause()` call to succeed. Therefore he is able to force the pausing of the auction contract at will.

Based on the gas requirements (1/64 of the gas calls has to be enough for `_pause()` gas cost of 21572), then `token.mint()` will need to consume at least 1359036 gas (63 * 21572), consequently it is only possible on some situations like founders with high percentage of vesting, for example 51 or more.

Consider the following POC. Here we are using another contract to restrict the gas limit of the call, but this can also be done with an EOA call from the attacker.

Exploit contract code:

```
pragma solidity ^0.8.16;

contract Attacker {
    function forcePause(address target) external {
        bytes4 selector =
↪   bytes4(keccak256("settleCurrentAndCreateNewAuction()"));
        assembly {
            let ptr := mload(0x40)
            mstore(ptr,selector)
```

```
            let success := call(1500000, target, 0, ptr, 4, 0, 0)
        }
    }
}
```

POC:

```
// SPDX-License-Identifier: MIT
pragma solidity 0.8.16;

import { NounsBuilderTest } from "./utils/NounsBuilderTest.sol";
import { MockERC721 } from "./utils/mocks/MockERC721.sol";
import { MockImpl } from "./utils/mocks/MockImpl.sol";
import { MockPartialTokenImpl } from "./utils/mocks/MockPartialTokenImpl.sol";
import { MockProtocolRewards } from "./utils/mocks/MockProtocolRewards.sol";
import { Auction } from "../src/auction/Auction.sol";
import { IAuction } from "../src/auction/IAuction.sol";
import { AuctionTypesV2 } from "../src/auction/types/AuctionTypesV2.sol";
import { TokenTypesV2 } from "../src/token/types/TokenTypesV2.sol";
import { Attacker } from "./Attacker.sol";

contract AuctionTest is NounsBuilderTest {
    MockImpl internal mockImpl;
    Auction internal rewardImpl;
    Attacker internal attacker;
    address internal bidder1;
    address internal bidder2;
    address internal referral;
    uint16 internal builderRewardBPS = 300;
    uint16 internal referralRewardBPS = 400;

    function setUp() public virtual override {
        super.setUp();
        bidder1 = vm.addr(0xB1);
        bidder2 = vm.addr(0xB2);
        vm.deal(bidder1, 100 ether);
        vm.deal(bidder2, 100 ether);
        mockImpl = new MockImpl();
        rewardImpl = new Auction(address(manager), address(rewards), weth,
↪   builderRewardBPS, referralRewardBPS);
        attacker = new Attacker();
    }

    function test_POC() public {
        // START OF SETUP
        address[] memory wallets = new address[](1);
        uint256[] memory percents = new uint256[](1);
```

```
        uint256[] memory vestingEnds = new uint256[](1);
        wallets[0] = founder;
        percents[0] = 99;
        vestingEnds[0] = 4 weeks;
        //Setting founder with high percentage ownership.
        setFounderParams(wallets, percents, vestingEnds);
        setMockTokenParams();
        setMockAuctionParams();
        setMockGovParams();
        deploy(foundersArr, tokenParams, auctionParams, govParams);
        setMockMetadata();
        // END OF SETUP

        // Start auction contract and do the first auction
        vm.prank(founder);
        auction.unpause();
        vm.prank(bidder1);
        auction.createBid{ value: 0.420 ether }(99);
        vm.prank(bidder2);
        auction.createBid{ value: 1 ether }(99);

        // Move block.timestamp so auction can end.
        vm.warp(10 minutes + 1 seconds);

        //Attacker calls the auction
        attacker.forcePause(address(auction));

        //Check that auction was paused.
        assertEq(auction.paused(), true);
    }
}
```

Impact

Should the conditions mentioned above be met, an attacker can arbitrarily pause the auction contract, effectively interrupting the DAO auction process. This pause persists until owners takes subsequent actions to unpause the contract. The attacker can exploit this vulnerability repeatedly.

Code Snippet

https://github.com/sherlock-audit/2023-09-nounsbuilder/blob/main/nouns-protocol/src/auction/Auction.sol#L238-L241

https://github.com/sherlock-audit/2023-09-nounsbuilder/blob/main/nouns-protocol/src/auction/Auction.sol#L292-L329

Tool used

Manual Review

Recommendation

Consider better handling the possible errors from `Token.mint()`, like shown below:

```
function _createAuction() private returns (bool) {
    // Get the next token available for bidding
    try token.mint() returns (uint256 tokenId) {
        //CODE OMMITED
    } catch (bytes memory err) {
        // On production consider pre-calculating the hash values to save gas
        if (keccak256(abi.encodeWithSignature("NO_METADATA_GENERATED()")) ==
↪   keccak256(err)) {
            _pause();
            return false
        } else if (keccak256(abi.encodeWithSignature("ALREADY_MINTED()") ==
↪   keccak256(err)) {
            _pause();
            return false
        } else {
            revert OUT_OF_GAS();
        }
    }
```

Discussion

nevillehuang

This is a previously known issue already proven to be not possible and incorrectly awarded in the previous audit as seen in the comments here.

Additionally, even if this is possible, DAO can unpause the auction FOC, meaning the attacker would be effectively losing funds from gas to maliciously pause the contract.

nevillehuang

Hi @neokry @Czar102 can you double check this issue just in case. There is some code changes that i missed out from the previous C4 audit that seems to indicate this issue is possible from the PoC. But I believe this to be still invalid/lowseverity given auction can be unpaused free of charged by the DAO and no further loss of funds since users cannot bid anyways given auction did not start so the DoS is not permanent, and the malicious user would effectively be wasting gas funds to pause the auction.

Arabadzhiew

Escalate

This actually seems to be a valid finding. This comment from the above linked issue from the previous C4 contest seems to explain the current situation quite well. The catch block was initially looking like this `catch Error(string memory) {`, while now it looks like this `catch {`. This means that initially, the catch block was only catching string errors, while now it will catch anything, including OOG reverts.

Additionally, since each unpause action will have to go through a separate governance proposal (which usually take a couple of days), this means that the auction participants will most likely lose interest in participating in the auctions of that particular DAO (especially if the auction gets paused multiple times), in turn - leading to a loss of funds for the DAO being exploited with this vulnerability.

Because of that, I believe that this issue warrants a ***Medium*** severity.

sherlock-admin2

> Escalate
>
> This actually seems to be a valid finding. This comment from the above linked issue from the previous C4 contest seems to explain the current situation quite well. The catch block was initially looking like this `catch Error(string memory) {`, while now it looks like this `catch {`. This means that initially, the catch block was only catching string errors, while now it will catch anything, including OOG reverts.
>
> Additionally, since each unpause action will have to go through a separate governance proposal (which usually take a couple of days), this means that the auction participants will most likely lose interest in participating in the auctions of that particular DAO (especially if the auction gets paused multiple times), in turn - leading to a loss of funds for the DAO being exploited with this vulnerability.
>
> Because of that, I believe that this issue warrants a ***Medium*** severity.

You've created a valid escalation!

To remove the escalation from consideration: Delete your comment.

You may delete or edit your escalation comment anytime before the 48-hour escalation window closes. After that, the escalation becomes final.

Oot2k

I completely agree with @Arabadzhiew escalation.

The C4 issue was actually invalid in the old codebase, but in the new current codebase its valid / a valid concern.

Please see #125 for a foundry POC which demonstrates the issue.

The impact of this is IMO medium, because it requires a specific set of admin params, which are less likely to accrue in real world. I still want to emphasis that this is exactly what the medium category is meant for.

For this reason I believe this issue and its duplicates are valid medium findings.

Best regards!

nevillehuang

1. No funds loss
2. DoS not permanent

For this reasons this should at most be low severity.

neokry

agree with @nevillehuang here. user also has to spend a relativly high amount of gas to execute this attack each time and realistically most DAOs have < 50% founder allocation

Arabadzhiew

Ok, first about the DOS part. Theoretically speaking, since in the `Governor` contract we have both the MAX_VOTING_DELAY and MAX_VOTING_PERIOD constants set to `24 weeks`, this means that if the `votingDelay` and `votingPeriod` values are set to those respective max values, a given auction can be paused for up to **336 days** at a time. Combining this with the fact that this exploit can be repeated more than once, and we see that this actually qualifies as a DOS issue, as per the Sherlock docs. Obviously, a lot of things have to line up perfectly in order for this to happen, but the point is that it can **theoretically** happen.

For the loss of funds part. I am just going to ask you all a question on this one. If you were going to participate in a NFT auction, but for some reason, this auction gets paused a number of times, for a couple of days each time, won't you just get fed up with that at some point and decide to go and invest your **assets** and time somewhere else?

nevillehuang

@Arabadzhiew Too many conditions need to be lined up for this to occur

1. Realistically speaking, no governance will set a voting period and voting delay of 24 weeks (5.5 months)
2. Even if theoretically possible, the attacker will need to perform this over and over again, each time wasting large amount of gas funds to block the auction, with no additional incentives to him

3. Since the auction has not started, no bidders can bid, so no-one will be losing funds here. No NFT will be minted to anyone, including the malicious attacker pausing.
4. The DAO can unpause at anytime by paying gas, so the cost of attack by the attacker is significantly larger than the DAO unpausing. So again, no incentives for him to perform this attack.
5. Sure users can get fed-up and governance may lose potential bidders, but I don't think that is a good enough reason to validate this issue as seen by this comment by @Czar102 here, where potential loss of profits from missing bidders is not a valid reason to validate the issue.

Arabadzhiew

I agree that both the impact from the DOS and the loss of fund issue are relatively questionable on their own, but given the fact that both of them are present with the vulnerability in question, it makes me consider it as one of a medium severity.

That's my take on the matter.

Czar102

What is the minimum percentage of the tokens founders need to have for this to be an issue? @Arabadzhiew From my understanding, prevention would be quite easy by simply minting the tokens as they are vested, but this wouldn't work if we already had an issue of an attacker submitting these low-gas transactions (if they were frontrunning). Also, can anyone be the attacker here, or only the founders?

From my understanding, it can classify as medium as stopping the auction impacts protocol's core protocol functionality. But the number of assumptions here may be quite large, so I am not sure if the assumptions are reasonable.

Arabadzhiew

> What is the minimum percentage of the tokens founders need to have for this to be an issue? @Arabadzhiew From my understanding, prevention would be quite easy by simply minting the tokens as they are vested, but this wouldn't work if we already had an issue of an attacker submitting these low-gas transactions (if they were frontrunning). Also, can anyone be the attacker here, or only the founders?
>
> From my understanding, it can classify as medium as stopping the auction impacts protocol's core protocol functionality. But the number of assumptions here may be quite large, so I am not sure if the assumptions are reasonable.

In this report it is stated that the minimum percentage is 51, but I can't confirm on that number. Also, yes, anyone can be an attacker here, since the

`settleCurrentAndCreateNewAuction` function, which is the one being used for this exploit, does not have any access control.

Oot2k

I am not sure but in the old report the amount needed was ~26. And yes technically the DOS can last a year / every time the dao its unpaused it can be dosed again. Most of the time proposal period is set to > 2 weeks so it would cut the earnings of a DAO significantly.

But yes the amount of founders tokens is quite unlikely, maybe if there is only a short auction time -> so high supply of tokens this could happen realistically.

Czar102

I believe this is a valid Medium severity issue on the grounds of impacting core protocol functionality (not DoS or loss of funds). Planning to accept the escalation.

As a side note, as @nevillehuang mentioned, opportunity loss of the DAO is not a loss of funds.

Czar102

Result: Medium Has duplicates

sherlock-admin2

Escalations have been resolved successfully!

Escalation status:

- Arabadzhiew: accepted

neokry

Fixed here: https://github.com/ourzora/nouns-protocol/pull/125

IAm0x52

Fix looks good. OOG errors during minting no longer pause the contract.

Issue M-2: MerkleReserveMinter minting methodology is incompatible with current governance structure and can lead to migrated DAOs being hijacked immediately

Source: https://github.com/sherlock-audit/2023-09-nounsbuilder-judging/issues/249

Found by

0x52, SilentDefendersOfDeFi, jerseyjoewalcott, nirohgo

Summary

MerkleReserveMinter allows large number of tokens to be minted instantaneously which is incompatible with the current governance structure which relies on tokens being minted individually and time locked after minting by the auction. By minting and creating a proposal in the same block a user is able to create a proposal with significantly lower quorum than expected. This could easily be used to hijack the migrated DAO.

Vulnerability Detail

MerkleReserveMinter.sol#L154-L167

```
unchecked {
    for (uint256 i = 0; i < claimCount; ++i) {
        // Load claim in memory
        MerkleClaim memory claim = claims[I];

        // Requires one proof per tokenId to handle cases where users want to
        ↪  partially claim
        if (!MerkleProof.verify(claim.merkleProof, settings.merkleRoot,
        ↪  keccak256(abi.encode(claim.mintTo, claim.tokenId)))) {
            revert INVALID_MERKLE_PROOF(claim.mintTo, claim.merkleProof,
            ↪  settings.merkleRoot);
        }

        // Only allowing reserved tokens to be minted for this strategy
        IToken(tokenContract).mintFromReserveTo(claim.mintTo, claim.tokenId);
    }
}
```

When minting from the claim merkle tree, a user is able to mint as many tokens as they want in a single transaction. This means in a single transaction, the supply of

the token can increase very dramatically. Now we'll take a look at the governor contract as to why this is such an issue.

Governor.sol#L184-L192

```
// Store the proposal data
proposal.voteStart = SafeCast.toUint32(snapshot);
proposal.voteEnd = SafeCast.toUint32(deadline);
proposal.proposalThreshold = SafeCast.toUint32(currentProposalThreshold);
proposal.quorumVotes = SafeCast.toUint32(quorum());
proposal.proposer = msg.sender;
proposal.timeCreated = SafeCast.toUint32(block.timestamp);

emit ProposalCreated(proposalId, _targets, _values, _calldatas, _description,
↪  descriptionHash, proposal);
```

Governor.sol#L495-L499

```
function quorum() public view returns (uint256) {
    unchecked {
        return (settings.token.totalSupply() * settings.quorumThresholdBps) /
        ↪  BPS_PER_100_PERCENT;
    }
}
```

When creating a proposal, we see that it uses a snapshot of the CURRENT total supply. This is what leads to the issue. The setup is fairly straightforward and occurs all in a single transaction:

1) Create a malicious proposal (which snapshots current supply)
2) Mint all the tokens
3) Vote on malicious proposal with all minted tokens

The reason this works is because the quorum is based on the supply before the mint while votes are considered after the mint, allowing significant manipulation of the quorum.

Impact

DOA can be completely hijacked

Code Snippet

MerkleReserveMinter.sol#L129-L173

Tool used

Manual Review

Recommendation

Token should be changed to use a checkpoint based total supply, similar to how balances are handled. Quorum should be based on that instead of the current supply.

Discussion

neokry

This is a valid issue but makes a big assumption that a malicious user is included in the merkle tree with a significant share of reserved tokens and the DAO has no veto set. This might be too much of an edge case to make the recommended changes to the token and governor contracts.

neokry

Fixed here: https://github.com/ourzora/nouns-protocol/pull/124

As noted in this PR we added a governance delay to fix the more significant issue of a DAO without veto potentially being hijacked. A delay will also help alleviate some of the issues related to claiming and quorum but not outright fix. For quorum issues we will warn DAOs around the risks of setting a high amount of votes for single users.

Oot2k

Escalate

I think this issue should be high instead of medium. The attack path can be simply executed by anyone and there is no way to prevent it.

The only requirement would be that there is no veto, but this is a feature not a requirement, which makes this a high instead of medium. Also see #155 for reasoning

As @nevillehuang mentions in the comment on #52 this should be more an High instead.

sherlock-admin2

> Escalate
>
> I think this issue should be high instead of medium. The attack path can be simply executed by anyone and there is no way to prevent it.

> The only requirement would be that there is no veto, but this is a feature not a requirement, which makes this a high instead of medium. Also see #155 for reasoning
>
> As @nevillehuang mentions in the comment on #52 this should be more an High instead.

You've created a valid escalation!

To remove the escalation from consideration: Delete your comment.

You may delete or edit your escalation comment anytime before the 48-hour escalation window closes. After that, the escalation becomes final.

neokry

> Escalate
>
> I think this issue should be high instead of medium. The attack path can be simply executed by anyone and there is no way to prevent it.
>
> The only requirement would be that there is no veto, but this is a feature not a requirement, which makes this a high instead of medium. Also see #155 for reasoning
>
> As @nevillehuang mentions in the comment on #52 this should be more an High instead.

the attack can only be executed by users in the merkle tree which is controlled by the admin. most DAOs also have veto set. there is no loss of funds if the merkle claim has no price per token and deployer can simply redeploy the DAO and remove the malicious user form the merkle tree.

nevillehuang

I see your point @Oot2k but I agree with sponsor, this issue is dependent on a NounsDAO type governance with no veto which I believe is unlikely, so will leave it up to @Czar102 to decide severity.

Czar102

Is the new DAO being created holding any funds? Or is it said that it shouldn't hold any until `settings.mintEnd`? @neokry @nevillehuang

neokry

DAOs are not holding any funds upon creation but it could receive funds if price per token is set and users have claimed tokens

nevillehuang

@Czar102 I only assigned this medium severity on the condition that veto are not set for a nounsDAO type governance which is unrealistic, but if you disagree, I

could also see why this could be high severity given the impact it has.

Oot2k

I wanted to add to sponsor comment that after migration the DAO indeed holds funds, the total layer 1 treasury.

So for example: DAO with 100 ETH is migrated, all old users get chance to mint tokens for free and no veto set because nouns is designed to delete the veto at some point.

In this case the 100 ETH can be instantly drained.

nevillehuang

@Oot2k Can you outline how the funds can be immediately drained? If true this definitely could be high severity.

From my understanding it has something to do with adjusting the parameters to allow instant execution of proposals correct?

Oot2k

In #155 we tried to outline it.

Migration happens like this:

1. DAO on layer 1 is stoped
2. Merkeltree with mint config is created
3. DAO on layer 2 is deployed
4. Treasury of DAO is transferred from layer1 to layer2
5. First user to mint quorum tokens can create proposal to drain DAO
6. Because of the nature of OZ governance only tokens minted at time of proposal can participate in governance. -> drain can not be stopped because only the malicious user can vote and there is no way to "save" treasury

nevillehuang

> In #155 we tried to outline it.
>
> Migration happens like this:
>
> 1. DAO on layer 1 is stoped
> 2. Merkeltree with mint config is created
> 3. DAO on layer 2 is deployed
> 4. Treasury of DAO is transferred from layer1 to layer2
> 5. First user to mint quorum tokens can create proposal to drain DAO

6. Because of the nature of OZ governance only tokens minted at time of proposal can participate in governance. -> drain can not be stopped because only the malicious user can vote and there is no way to "save" treasury

Is there a proposal delay/duration that can be changed by the malicious users? If so it will be helpful if you point me to the code logic. If not I think the other members of the DAO would have sufficient time to react to proposals.

neokry

The treasury migration step has to go through the full layer 1 DAO governance and the DAO can decide when to submit that proposal. it doesn't happen immediately after the L2 DAO is deployed. An L1 DAO might also choose to only migrate a portion of their treasury as well to ensure the L2 DAO runs smoothly for a period of time

Oot2k

> In #155 we tried to outline it. Migration happens like this:
>
> 1. DAO on layer 1 is stoped
> 2. Merkeltree with mint config is created
> 3. DAO on layer 2 is deployed
> 4. Treasury of DAO is transferred from layer1 to layer2
> 5. First user to mint quorum tokens can create proposal to drain DAO
> 6. Because of the nature of OZ governance only tokens minted at time of proposal can participate in governance. -> drain can not be stopped because only the malicious user can vote and there is no way to "save" treasury
>
> Is there a proposal delay/duration that can be changed by the malicious users? If so it will be helpful if you point me to the code logic. If not I think the other members of the DAO would have sufficient time to react to proposals.

They cant react. OZ governance takes the votes from the timestamp. This report includes all important code snippets. Even if the proposal has an execution time of 10 weeks it does not matter because every proposal created after gets executed after the drain.

Sponsors comments are right, ofc the DAO can migrate only a part of funds, but that does not lower the severity. Its still highly realistic that all funds are send at ones.

And in reality I think no one will claim tokens in a DAO that has no treasury, so the possibility is quite high.

neokry

for full context on migration we have a bot setup to airdrop DAO tokens for migrated DAOs. Our recommendation to all DAOs will be to execute the migration call wait for tokens to be airdropped and pass a proposal to unpause auctions on L2 before sending the treasury to the L2 DAO. there is a chance this attack could be executed before the bot airdrops the tokens which is why we've added the governance delay as a fix. also a malicious user would need to immediately submit this proposal to the DAO making it obvious if a DAO will be captured ie they can easily choose not to send the treasury.

Czar102

I think sending any DAO funds to the L2 DAO would be irresponsible if the L2 DAO isn't set up yet. I would be considered an admin error in my opinion. @neokry would you agree?

> DAOs are not holding any funds upon creation but it could receive funds if price per token is set and users have claimed tokens

I think mint fees could be stolen. The attacker could wait for as many tokens as possible would be bought and then mint all their tokens (so that they will have a majority and satisfy quorum) and create a proposal to steal all mint fees. From my understanding, they can also hijack the governance, but it wouldn't hold any other funds at that time.

Is my understanding accurate? I would also like to get to know why do you perceive having no veto as likely/unlikely.

nevillehuang

@Czar102 I am basing it off of the original NounsDAO governance, where veto is a core role present configured by the admin to prevent malicious proposals.

neokry

I agree on your first point @Czar102 . regarding stealing mint fees the attack only works if the attackers vote power is greater than the voting power of active voters. ie if they wait for the majority of users to claim their mint funds they have a higher chance of their proposal to steal funds be voted down.

also agree with @nevillehuang while veto is optional the veto is used by most DAOs and we strongly encouraged DAOs to set it up to prevent governance attacks like this.

Czar102

My thoughts after some internal discussions:

- This issue presents a loss of funds scenario, which is limited by the number of tokens an attacker can gather (when there is no frontrunning, frontrunning also has a limited impact). This is quite constrained, so is the loss. The DAO shouldn't hold any funds other than buy-ins at that time.
- If there is no vetoer and the attacker does the attack right, I believe there is no way to recover funds for addresses who bought in.
- It is reasonable to assume that there will be no vetoer because of this fragment of the code, hence this finding should be at least a medium. It feels like that role is highly advisable to be assigned at least in the very first moments of the DAO when it's being set up (and tokens are being bought, etc.), so I believe a lack of the vetoer is an assumption concerning the state of the DAO.

Based on the above points, I think it is a borderline Med/High. I am leaning towards leaving it a medium.

Oot2k

I want to quote the scenario where a DAOs treasury is transferred before the reserve mint ends. The code/docs do not mention in any way that the mint of tokens happens before the treasury is transferred. In this case the impact is detrimental. I still think this is a high considering the different ways the issue impacts governance manipulation.

Czar102

I think it would be a setup mistake to send funds to an "uninitialized" DAO.

Czar102

Result: Medium Has duplicates

sherlock-admin2

Escalations have been resolved successfully!

Escalation status:

- Oot2k: rejected

IAm0x52

Fix looks good. A governance delay has been added to the governor contract. Proposals can only be created after either 1) all reserve tokens have been minted or 2) the delay has passed.

Issue M-3: Migration transactions executed out of order can lock DAOs

Source: https://github.com/sherlock-audit/2023-09-nounsbuilder-judging/issues/326

Found by

Protocol Team

Summary

The following migration calls have the potential to be executed out of order locking the new L2 DAO:

- `deploy`
- `callMetadataRenderer`
- `renounceOwnership`

ie if a DAO is deployed and then ownership is immediately renounced without the required metadata renderer calls the DAO will have no metadata set and any `Token.mint` calls will revert with a metadata error.

a malicious actor can watch for `deploy` calls and block any migrated DAOs by immediately relaying the `renounceOwnership` call before any `callMetadataRenderer` calls.

this issue occurs because the `L2CrossDomainMessenger` has no sense of ordering and the `relayMessage` calls can be called in any order. as long as `deploy` has been called the `renounceOwnership` call will succeed.

Discussion

neokry

Fixed here: https://github.com/ourzora/nouns-protocol/pull/128

IAm0x52

Fix looks good. renounceOwnership will now revert if the proper number of metadata calls are not relayed prior.