yAudit NounsDAO Token Buyer Review

Review Resources:

- Code Repo,
- Spec

Auditors:

- NibblerExpress
- blockdev

Table of Contents

Review Summary

NounsDAO Token Buyer

TokenBuyer's purpose is to allow the NounsDAO to pay the proposals with ERC20 tokens. Since trading a large portion of ETH incurs slippage and is susceptible to sandwich attacks, this protocol uses Chainlink oracle to fetch ETH prices and allow anyone to sell their tokens against ETH.

The contracts of Token Buyer Repo were reviewed over 7 days, 1 day of which was used to create an initial overview of the contract. The code review was performed between October 11, 2022 and October 18, 2022. The code was reviewed by 2 auditors for a total of 32 review hours (NibblerExpress 8 hours, blockdev 24 hours). The review was limited to the latest commit at the start of the review. This was commit `23d64ac7093f504ad4731bc4cf8d41b2c2943657` for the `token-buyer` repo.

Scope

The scope of the review consisted of the following contracts at the specific commit:

- TokenBuyer.sol
- PriceFeed.sol
- Payer.sol

After the findings were presented to the NounsDAO team, fixes were made and included in several PRs.

This review is a code review to identify potential vulnerabilities in the code. The reviewers did not investigate security practices or operational security and assumed that privileged accounts could be trusted. The reviewers did not evaluate the security of the code relative

to a standard or specification. The review may not have identified all potential attack vectors or areas of vulnerability.

yAudit and the auditors make no warranties regarding the security of the code and do not warrant that the code is free from defects. yAudit and the auditors do not represent nor imply to third parties that the code has been audited nor that the code is free from defects. By deploying or using the code, NounsDAO and users of the contracts agree to use the code at their own risk.

Code Evaluation Matrix

Category	Mark	Description
Access Control	Good	Privileged functionality is restricted to the owner and admin.
Mathematics	Good	Decimal adjustment is done to convert between Oracle price and wad decimal.
Complexity	Good	Responsibilities are separated which reduces complexity.
Libraries	Good	Queue library from Open Zeppelin is modified to fit the need, others are imported.
Decentralization	Good	Core protocol functions are permissionless.
Code stability	Good	Changes were reviewed at a specific commit hash and the scope was not expanded after the review was started. The code reviewed had all features implemented.
Documentation	Good	Descriptive NatSpec comments are found in the interface contracts. The comments accurately describe the function. The spec contains helpful information about the intended use and purpose.
Monitoring	Average	Events are used.
Testing and verification	Good	Forge tests have high coverage and a deploy script is used.

Findings Explanation

Findings are broken down into sections by their respective impact:

- Critical, High, Medium, Low impact
  - These are findings that range from attacks that may cause loss of funds, impact control/ownership of the contracts, or cause any unintended consequences/actions that are outside the scope of the requirements
- Gas savings
  - Findings that can improve the gas efficiency of the contracts
- Informational
  - Findings including recommendations and best practices

High Findings

1. High - Price range should be within Chainlink's range (blockdev, NibblerExpress)

`PriceFeed` is already deployed at `0x4050Cd1eDDB589fe26B62F8859968cC9a415cE7F` according to the deploy script. `priceLowerBound` is set to $100 * 10^{18}$, and `priceUpperBound` is set to $100,000 * 10^{18}$ which translate to $100 and $100K respectively.

This is the Chainlink oracle `PriceFeed` is using: `0x5f4eC3Df9cbd43714FE2740f5E3616155c5b8419` which uses this aggregator: `0x37bC7498f4FF12C19678ee8fE19d713b87F6a9e6`. `minAnswer` and `maxAnswer` values from this aggregator are: $1 * {10^8}$ and $10,000 * {10 ^ 8}$ which translate to $1 and $10K. Chainlink will never report prices outside this range.

Hence, the upper bound of $100K set in `PriceFeed` is not useful as the check `priceWad > priceUpperBound` is always `false`:

```
if (priceWAD < priceLowerBound || priceWAD > priceUpperBound) {
    revert InvalidPrice(priceWAD);
}
```

Technical Details

PriceFeed.sol#L93

Impact

High. If the market price moves above the upper bound specified by Chainlink, `TokenBuyer` will allow trades in the loss for Nouns DAO until the staleness check kicks in. `staleAfter` is set to 1 hour currently.

Recommendation

Set `priceUpperBound` to a value < $10,000 * 10^{18}$. Chainlink reports a new price in two scenarios:

1. The price moves beyond a certain deviation threshold.
2. The price has not been reported within a certain amount of time (aka heartbeat).

Suppose you set your upper bound to $9K, and the market price movement is say from point $8K to $11K.

There are 2 scenarios:

1. the price directly jumped from $8K to $11K, then the upper bound won't be useful and only the staleness check will protect from trading. In this case, `priceUpperBound` is useless in both cases (in the current code and the recommendation).
2. the price moves from $8K -> $9.5K -> $11K, in this case, `priceUpperBound` will prevent `TokenBuyer` from making any trades. However, the current code will let it trade at $9.5K until the staleness check comes in. And this is a more likely scenario as Chainlink reports a price quickly when it moves above a certain deviation.

Developer Response

Our current path forward will be to set the cap at $8K, and in the future upgrade the price feed to check the diff between Chainlink and Univ3 TWAP and revert in case of a big difference.

Medium Findings

1. Medium - ETH buyer pays the gas cost of debt payments (blockdev)

`buyETH()` function calls `payBackDebt()` which iterates over the queue and pays the debt of the first few elements. Iterating over the queue is a gas-intensive operation. If the queue is

long relative to the buy amount, it may discourage the buyers as the extra gas costs may overshadow the discount.

Technical Details

TokenBuyer.sol#L195, TokenBuyer.sol#L235

Impact

Medium. The extra gas costs may disincentivize trading through `TokenBuyer`.

Recommendation

Consider removing the debt payment in `buyETH()` functions. Anyone can call `payBackDebt()` separately for debt payments.

Note: With this change, you can also consider calling `payBackDebt()` at the beginning of `sendOrRegisterDebt()` since the debt in the queue should be paid first.

Developer Response

Acknowledged, won't fix.

We are aware of this issue and prefer to use it as is in order to ensure better UX by not needing anyone to manually call `payBackDebt()`.

We expect the queue to remain short at all times. In case the queue becomes longer, we can adjust the discount parameter to increase incentives for trading.

Low Findings

1. Low - Can require the basis point values to be less than `10_000` (blockdev)

`TokenBuyer` takes several constructor arguments which are denominated in basis points. Constructor can require them to be < `10_000` to be safe.

Technical Details

TokenBuyer.sol#L146-L148

Impact

Low. `price()` will return incorrect values in this case, however since the likelihood of this happening is low, the overall impact is limited.

Recommendation

Consider either:

- Adding `require` checks in `TokenBuyer` constructor to assert that all the basis point values are < `10_000`.
- Adding these `require` checks in forge script.

Developer Response

Fixed in commit 5e252d6.

2. Low - Use a two-step Ownership transfer pattern (blockdev)

If a wrong address is passed to the function `transferOwnership()`, privileged functionality is permanently lost.

Technical Details

TokenBuyer.sol, Payer.sol

Impact

Low.

Recommendation

Consider implementing a two-step ownership transfer where the proposed new owner has to explicitly accept the role. OpenZeppelin now provides this functionality via `Ownable2Step.sol`

Developer Response

Acknowledged, won't fix.

We intend for the owner of the contracts to be the nouns timelock, so any ownership transfer will go through a proposal process with ample time to check for errors. In addition, when we transfer the ownership to the DAO the first time, we don't want to go through a proposal process in order to accept the ownership.

3. Low - Existing debts should be prioritized in `sendOrRegisterDebt()` (blockdev)

`sendOrRegisterDebt(account, amount)` uses current token balance to pay `account`. In the current design, any new balance bought through `TokenBuyer` pays the existing debt. So if `Payer` has a balance, it means that all the existing debt was cleared, or the payment token was directly transferred to it. In the second case, the token balance is used to pay `account`

bypassing the existing debt in the queue.

Technical Details

Payer.sol#L100

Impact

Low. Since the existing debt takes priority over the new debt, it should be paid first. It may also be seen as impartiality towards a party.

Recommendation

`sendOrRegisterDebt()` can first call `payBackDebt()` and then proceed to pay the current account with the remaining amount.

Developer Response

Acknowledged, won't fix.

We don't plan to send funds to the `Payer` not through the `TokenBuyer`. In case it does happen, we can manually trigger `payBackDebt()`.

4. Low - Risk of USDC depeg (blockdev)

`PriceFeed` uses Chainlink oracle of ETH/USD for payment token USDC. This assumes that USDC is always pegged to $1. In the case that the USDC price moves below $1, `TokenBuyer` will make trades at a loss.

Technical Details

DeployTokenBuyer.s.sol

Impact

Low. USDC depegging can be considered an unlikely event. However, in case that happens, all the ETH locked in `TokenBuyer` will be making trades at a loss.

Recommendation

Consider using Chainlink's USDC/ETH oracle. Another option is to monitor USDC price off-chain and pause the contract in case it depegs. However, the risk is that the pause transaction can be frontrun by trade transactions.

Developer Response

Acknowledged, won't fix.

We prefer to use the ETH/USD oracle because of the significantly shorter heartbeat (1 hr vs 24 hrs). We are aware of the depeg risk and consider it very low for USDC.

5. Low - Fee-on-transfer token not supported (blockdev)

The payment token is assumed to send the full amount to the receiver on a transfer. However, if a token like USDT turns on a fee in the future, it can break accounting logic.

Technical Details

TokenBuyer.sol

Impact

Low. The payment token is defined by the deployer. The deployer can ensure the payment token does not and cannot take a fee on transfer (like USDT). The current code is fine for such an ERC20. However, if a fee-on-transfer ERC20 is used as the payment token, the code should be updated to handle the fee.

Recommendation

Consider one of the following two approaches:

- Either ensure off-chain before deploying that the payment token is not a fee-on-transfer token.
- Update the contract logic to calculate the tokens transferred by checking the token balance before and after a transfer.

Developer Response

Acknowledged, won't fix.

We don't plan to use any tokens with fee-on-transfer in the near future. If we do, we will revise the logic.

Gas Savings Findings

1. Gas - Replace `owner()` with `msg.sender` in `withdrawPaymentToken()` (blockdev)

In `onlyOwner` functions, `owner()` is always equal to `msg.sender`, otherwise it reverts. In these cases, using `msg.sender` instead of `owner()` saves gas since it prevents expensive storage read.

Technical Details

Payer.sol#L123

Impact

Gas savings.

Recommendation

Replace `owner()` with `msg.sender`:

```
function withdrawPaymentToken() external onlyOwner {
-   address to = owner();
+   address to = msg.sender;
```

Developer Response

Fixed in commit 906a3d3.

2. Gas - Cache `totalDebt` in `payBackDebt()` (blockdev)

In each iteration of the loop in `payBackDebt()`, the storage variable `totalDebt` is updated. Updating it only once at the end is a cheaper alternative since it updates storage only once.

Technical Details

Payer.sol#L158, Payer.sol#L173

Impact

Gas savings.

Recommendation

Consider doing the following:

1. Cache `totalDebt` at the beginning.
2. Accumulate total debt paid in a separate variable.
3. `break` instead of returning in the `if` condition.
4. Update `totalDebt` at the end.

Developer Response

Fixed in commit a4c601e.

3. Gas - Use `_debtAmount` instead of `debt.amount` (blockdev)

Avoiding storage reads saves gas. Here, `debt.amount` can be replaced with `_debtAmount`.

```
uint96 _debtAmount = debt.amount;
address _debtAccount = debt.account;

if (amount < _debtAmount) {
    // Not enough to cover entire debt, pay what you can and leave
    // cast is safe because `amount` < `_debtAmount` (uint96)
    uint96 remainingDebt = debt.amount - uint96(amount);
```

Technical Details

Payer.sol#L146-L152

Impact

Gas savings.

Recommendation

Apply this diff:

```
-uint96 remainingDebt = debt.amount - uint96(amount);
+uint96 remainingDebt = _debtAmount - uint96(amount);
```

Developer Response

Fixed in commit 14343d9.

4. Gas - `DebtQueue.empty()` can just check for equality (blockdev)

Unless `_end` overflows, `_begin <= _end` is an invariant. `DebtQueue` also mentions this in a comment:

```
// The interface preserves the invariant that begin <= end
```

Hence,

```
function empty(DebtDeque storage deque) internal view returns (bool) {
```

```
    return deque._end <= deque._begin;
}
```

can be gas optimized by just checking `_end == _begin`.

Technical Details

DebtQueue.sol#L112

Impact

Gas savings.

Recommendation

Apply this diff:

```
function empty(DebtDeque storage deque) internal view returns (bool) {
-    return deque._end <= deque._begin;
+    return deque._end == deque._begin;
}
```

Developer Response

Acknowledged, won't fix.

To our understanding, the gas optimization here is very minor. We prefer to make less changes to the OZ Queue lib for such small gas gain.

5. Gas - Precompute decimal factor to save gas and avoid magic numbers (NibblerExpress)

The function `ethAmountPerTokenAmount()` calculates `((tokenAmount * 1e36) / price()) / paymentTokenDecimalsDigits`. The function could save gas and avoid using the magic number 1e36 by having the constructor precompute a decimal factor equal to Eth decimals * price decimals / payment token decimals (i.e., 1e36/paymentTokenDecimalsDigits). The factor could also be used in `tokenAmountPerEthAmount()`. This should be fine for USDC which only has 6 decimals. However, for payment tokens with high decimals, this can lead to precision loss.

Technical Details

TokenBuyer.sol#L292, TokenBuyer.sol#L323

Impact

Gas savings.

Recommendation

Depending on the future plan, if only USDC is being used as a payment token, precompute the decimal factor as discussed above.

Developer Response

Acknowledged, won't fix.

We prefer to keep the implementation agnostic to the number of decimals used in the token and not make any assumptions.

Informational Findings

1. Informational - `TokenBuyer` may not use the entire received amount to pay the debt (blockdev)

`buyETH(tokenAmount, to, data)` transfers `tokensReceived` amount of payment token to `payer`, but only uses `amount` to pay debt. If `tokensReceived > amount`, more debt can be paid if `tokenReceived` is used. Additionally, the event emitted is incorrect:

```
_payer.payBackDebt(amount);
emit SoldETH(to, ethAmount, amount);
```

Technical Details

TokenBuyer.sol#L235-L237

Impact

Informational. Anyone can call `payBackDebt()`, so even if a lower amount is used it just takes one more transaction to pay debt using the remaining received tokens. In reality, the extra tokens received will likely be dust.

Recommendation

Apply this diff to line 235:

```
-_payer.payBackDebt(amount);
+_payer.payBackDebt(tokensReceived);
```

```
-emit SoldETH(to, ethAmount, amount);
+emit SoldETH(to, ethAmount, tokenReceived);
```

Developer Response

Fixed in commit 96a3fb0.

2. Informational - `TokenBuyer`'s constructor can fetch `paymentToken` from `payer` (blockdev)

`TokenBuyer`'s constructor takes `paymentToken` as an argument. However, it can be fetched from `payer` which is another argument. This prevents deployment errors.

Technical Details

TokenBuyer.sol#L153

Impact

Informational. If deployment is correct, this won't be an issue.

Recommendation

Fetch `paymentToken` info from `payer` in constructor:

```
paymentToken = payer.paymentToken();
```

Developer Response

Fixed in commit 9f744fb.

3. Informational - `DebtDeque`'s `_begin` and `_end` can be `uint128` (blockdev)

`DebtDeque`'s Natspec says:

```
Indices are signed integers because the queue can grow in any direction.
```

However, it isn't possible to have negative indices if `_end` is initially `0`. `_begin` and `_end` are always incremented by `1` via `popFront()` and `pushBack()`, and the assumption is that this won't overflow:

```
* Since the items are added one at a time we can safely
* assume that these 128-bit indices will not overflow, and use unchecked arithmetic.
```

Technical Details

DebtQueue.sol#L30-L44

Impact

Informational.

Recommendation

Use `uint128` for `_begin`, `_end`, and `_data` key.

Developer Response

Acknowledged, won't fix.

Since we don't gain much from the suggested change, we prefer to keep it close to the original OZ Queue code.

4. Informational - Review tokens before supporting them via `TokenBuyer` (blockdev)

NounsDAO team has indicated that tokens like USDC or DAI are planned to be supported via this system. However, if more tokens are planned like ERC-777, or fee-on-transfer tokens, the team should review the implementation to make sure the system remains secure.

This issue does not apply to payment tokens like USDC, DAI, WETH. However, functions like `sendOrRegisterDebt()` and `payBackDebt()` are open for reentrancy via `safeTransfer()` if other tokens are considered.

Technical Details

Repo

Impact

Informational. For stablecoins like USDC and DAI, this issue does not surface any security vulnerability.

Recommendation

Consider reviewing any new payment token being considered for reentrancy attacks and

any fee. Also, note the issue “Fee-on-transfer token not supported”.

Developer Response

Acknowledged.

5. Informational - `onlyAdmin()` is not used (NibblerExpress)

The modifier `onlyAdmin()` is not used. The modifier `onlyAdminOrOwner()` is used for all functions the admin can call.

Technical Details

TokenBuyer.sol#L120-L125

Impact

Informational.

Recommendation

Delete the onlyAdmin() modifier and error.

Developer Response

Fixed in commit 05839da.

6. Informational - Consider solidity 0.8.16 or higher (NibblerExpress)

Solidity 0.8.16 or 0.8.17 is currently recommended over 0.8.15.

Technical Details

Example: TokenBuyer.sol#L16

Impact

Informational. Vulnerabilities in 0.8.15 do not appear to be present in the code.

Recommendation

Use solidity 0.8.16 or higher.

Developer Response

Fixed in commit 09c7101.

7. Informational - Consider adding events for creating and paying back the debt if the debt is paid back on creation (invader-tak)

The function `sendOrRegisterDebt()` in the `Payer.sol` contract can under certain

circumstances get and pay back a debt during its call - presumably, this is done to save the gas to register the debt and calling the payBackDebt function - This may cause complications for data collection, as no `RegisteredDebt` or `PaidBackDebt` events are emitted, especially when using services such as the Graph or Dune analytics.

Technical Details

Payer.sol#L100

Impact

Informational. Streamline data collection for protocol.

Recommendation

Add a `RegisteredDebt` and `PaidBackDebt`, or a separate event for a registered debt that gets paid back during registration.

Developer Response

We are assuming we can use ERC20 transfer events to track flow of funds, and use the debt related events just for cases when funds are delayed which can be helpful for tracking things like the time it takes to pay back debt or how often we are in debt.

Final remarks

blockdev

Responsibilities are split in different contracts, and the code is well documented making it easy to understand.

NibblerExpress

The code allows any user to swap payment tokens for ETH, which has a high risk. Using pausable, nonreentrant, and Chainlink oracles mitigate much of the risk. Overall, the code is well organized and commented on.
