Rolling Dutch Auction Security Review

Pashov Audit Group

Conducted by: pashov

April 8th, 2023

Contents

1. About pashov

Krum Pashov, or **pashov**, is an independent smart contract security researcher. Having found numerous security vulnerabilities in various protocols, he does his best to contribute to the blockchain ecosystem and its protocols by putting time and effort into security research & reviews. Check his previous work here or reach out on Twitter @pashovkrum.

2. Disclaimer

A smart contract security review can never verify the complete absence of vulnerabilities. This is a time, resource and expertise bound effort where I try to find as many vulnerabilities as possible. I can not guarantee 100% security after the review or even if the review will find any problems with your smart contracts. Subsequent security reviews, bug bounty programs and on-chain monitoring are strongly recommended.

3. Introduction

A time-boxed security review of the **Rolling Dutch Auction** protocol was done by **pashov**, with a focus on the security aspects of the application's smart contracts implementation.

4. About Rolling Dutch Auction

The Rolling Dutch Auction protocol is a dutch auction derivative with composite decay. This means that the auction will start off with a high asking price and it will gradually be decreased with time. The protocol works by auctioning ERC20 tokens that can be bid on with other ERC20 tokens themselves. Both the auction and bid implementations are custodial, meaning the protocol is transferring and holding ERC20 funds in itself.

The way the auction works is by having windows, which are periods where the current price is frozen. Anytime a bid is submitted while a window is active, the duration of the window is restarted, meaning it implements a perpetual duration mechanism. When a window expires the current selling price is referenced by calculating the time elapsed from the last window instead of the auction's initiation.

Unexpected/Interesting Design choices

The `withdraw` functionality is callable by anyone, it will transfer the proceeds and the remaining reserves to the `operatorAddress`. Same for `redeem`, anyone can call it and it will transfer the refund and claim amount to the `bidder` argument.

To overbid another bidder, you have to place a bid that has both a higher `price` and higher `volume` than the preceding highest bid.

Both the `scalarPrice` method and the `price` parameter in `commitBid` are valued in terms of the `reserve` token.

Threat Model

Roles & Actors

- Auction creator - an account that calls `createAuction` and loads the initial reserve into the `RDA` contract
- Auction operator - the account that will receive the proceeds & remaining reserves when `withdraw` is called
- Bidder - an account that submits a bid via the `commitBid` functionality, moving his `purchaseToken` funds into the `RDA` contract
- Unauthorized user - anyone can call `fulfillWindow`, `withdraw` and `redeem` by just paying gas

Security Interview

Q: What in the protocol has value in the market?

A: The protocol is custodial, so the purchase and reserve tokens amounts it holds are valuable.

Q: What is the worst thing that can happen to the protocol?

A: The funds it holds get stuck in it forever or a user steals them, making other users' redeems/withdraws revert.

Q: In what case can the protocol/users lose money?

A: If users receive less (or none at all) reserve tokens than what they bid for. Same for the protocol operator - if he receives less (or none at all) purchase tokens than what he should have.

Potential attacker's goals

- Place any method in the protocol into a state of DoS
- Steal another user's claimable reserve tokens
- Exploit bugs in price calculations

Potential ways for the attacker to achieve his goals

- Making calls to `createAuction`, `commitBid`, `redeem` and `withdraw` revert by force-failing a `require` statement or an external call
- Exploit errors or rounding downs in divisions in price calculations for personal benefit
- Force the auction to never complete so no one will receive tokens

5. Risk Classification

Severity	Impact: High	Impact: Medium	Impact: Low
Likelihood: High	Critical	High	Medium
Likelihood: Medium	High	Medium	Low
Likelihood: Low	Medium	Low	Low

5.1. Impact

- High - leads to a significant material loss of assets in the protocol or significantly harms a group of users.
- Medium - only a small amount of funds can be lost (such as leakage of value) or a core functionality of the protocol is affected.
- Low - can lead to any kind of unexpected behavior with some of the protocol's functionalities that's not so critical.

5.2. Likelihood

- High - attack path is possible with reasonable assumptions that mimic on-chain conditions, and the cost of the attack is relatively low compared to the amount of funds that can be stolen or lost.
- Medium - only a conditionally incentivized attack vector, but still relatively likely.
- Low - has too many or too unlikely assumptions or requires a significant stake by the attacker with little or no incentive.

5.3. 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

6. Security Assessment Summary

***review commit hash* - cb27022597db95c4fd734356491bc0304e1e0721**

Scope

The following smart contracts were in scope of the audit:

- `RDA`
- `interfaces/IRDA`

7. Executive Summary

Over the course of the security review, pashov engaged with Rolling Dutch Auction to review Rolling Dutch Auction. In this period of time a total of **17** issues were uncovered.

Protocol Summary

Protocol Name	Rolling Dutch Auction
Date	April 8th, 2023

Findings Count

Severity	**Amount**
Critical	3
High	1
Medium	4
Low	3
QA	6
Total Findings	**17**

Summary of Findings

ID	Title	Severity	Status
[C-01]	Anyone can make new bids always revert after a window expires	Critical	Resolved
[C-02]	Successful bidders can lose significant value due to division rounding	Critical	Resolved
[C-03]	The logic in elapsedTime is flawed	Critical	Resolved
[H-01]	Users are likely to lose their bid if purchaseToken is a low-decimals token	High	Resolved
[M-01]	Missing input validation on createAuction function parameters can lead to loss of value	Medium	Resolved
[M-02]	Loss of precision in scalarPrice function	Medium	Resolved
[M-03]	Protocol won't work correctly with tokens that do not revert on failed transfer	Medium	Resolved
[M-04]	Auction won't work correctly with fee-on-transfer & rebasing tokens	Medium	Resolved
[L-01]	Auction with price == 0 can be re-created	Low	Resolved
[L-02]	The commitBid method does not follow Checks-Effects-Interactions pattern	Low	Resolved
[L-03]	The scalarPrice method should have an activeAuction modifier	Low	Resolved
[QA-01]	Using require statements without error strings	QA	Resolved
[QA-02]	Typos in code	QA	Resolved
[QA-03]	Missing License identifier	QA	Resolved

[QA-04]	Function state mutability can be restricted to view	QA	Resolved
[QA-05]	Incomplete NatSpec docs	QA	Resolved
[QA-06]	Missing override keyword	QA	Resolved

8. Findings

8.1. Critical Findings

[C-01] Anyone can make new bids always revert after a window expires

Severity

Impact: High, as all new bidding will revert until auction ends

Likelihood: High, as anyone can execute the attack without rare preconditions

Description

The `fulfillWindow` method is a `public` method that is also called internally. It sets `window.processed` to `true`, which makes it callable only once for a single `windowId`. The problem is that the `commitBid` function has the following logic:

```
if (hasExpired) {
    window = _window[auctionId][windowExpiration(auctionId)];
}
```

Where `windowExpiration` calls `fulfillWindow` with the latest `windowId` in itself. If any user manages to call `fulfillWindow` externally first, then the `window.processed` will be set to `true`, making the following check in `fulfillWindow`

```
if (window.processed) {
    revert WindowFulfilled();
}
```

revert on every `commitBid` call from now on. This will result in inability for anyone to place more bids, so the auction will not sell anything more until the end of the auction period.

Recommendations

Make `fulfillWindow` to be `internal` and then add a new public method that calls it internally but also has the `inactiveAuction` modifier as well - this way anyone will be able to complete a window when an auction is finished even though no one can call `commitBid`.

[C-02] Successful bidders can lose significant value due to division rounding

Severity

Impact: High, as possibly significant value will be lost

Likelihood: High, as it will happen with most bids

Description

The `fulfillWindow` method calculates the auction reserves and proceeds after a successful bid in a window. Here is how it accounts it in both the `auctions` and `claims` storage mappings:

```
_auctions[auctionId].reserves -= volume / price;
_auctions[auctionId].proceeds += volume;

_claims[bidder][auctionId] = abi.encode(refund - volume, claim +
  (volume / price));
```

The problem is in the `volume / price` division and the way Solidity works - since it only has integers, in division the result is always rounded down. This would mean the bidder will have less `claim` tokens than expected, while the `_auctions[auctionId].reserves` will keep more tokens than it should have. Let's look at the following scenario:

1. The `reserve` token is `WETH` (18 decimals) and the purchase token is `DAI` - 18 decimals as well
2. Highest bidder in the window bid 2 * 1e18 - 1 `DAI` with a price of 1e18 `WETH`
3. While the bid should have resulted in 1.99999 `WETH` bought, the user will receive only 1 `WETH` but not get a refund here

4. The user got the same amount of `WETH` as if he bid 1e18 but he bid twice as much, minus one

Every remainder of the `volume / price` division will result in a loss for the bidder.

Recommendations

Design the code so that the remainder of the `volume / price` division gets refunded to the bidder, for example adding it to the `refund` value.

[C-03] The logic in `elapsedTime` is flawed

Severity

Impact: High, as the method is used to calculate the price of the auction but will give out wrong results

Likelihood: High, as the problems are present almost all of the time during an auction

Description

There are multiple flaws with the `elapsedTime` method:

1. If there are 0 windows, the `windowIndex` variable (which is used for the windows count) will be 1, which is wrong and will lead to a big value for `windowElapsedTime` when it should be 0
2. If `auctionElapsedTime == windowElapsedTime` we will get `auctionElapsedTime` as a result, but if there was just 1 more second in `auctionElapsedTime` we would get `auctionElapsedTime - windowElapsedTime` which would be 1 as a result, so totally different result
3. When a window is active, the `timestamp` argument will have the same value as the `auction.startTimestamp` so `auctionElapsedTime` will always be 0 in this case

The method has multiple flaws and works only in the happy-case scenario.

Recommendations

Remove the method altogether or extract two methods out of it, removing the `timestamp` parameter to simplify the logic. Also think about the edge case scenarios.

8.2. High Findings

[H-01] Users are likely to lose their bid if `purchaseToken` is a low-decimals token

Severity

Impact: High, because users will lose their entire bid amount

Likelihood: Medium, because it happens when `purchaseToken` is a low-decimals token, but those are commonly used

Description

When a user calls `commitBid` he provides a `volume` parameter, which is the amount of `purchaseToken` he will bid, and a `price` parameter, which is the price in `reserveToken`. His bid is then cached and when window expires the `fulfillWindow` method is called, where we have this logic:

```
_auctions[auctionId].reserves -= volume / price;
_auctions[auctionId].proceeds += volume;

_claims[bidder][auctionId] = abi.encode(refund - volume, claim +
  (volume / price));
```

The problem lies in the `volume / price` calculation. In the case that the `reserveToken` is a 18 decimal token (most common ones) but the `purchaseToken` has a low decimals count - `USDC`, `USDT` and `WBTC` have 6 to 8 decimals, then it's very likely that the `volume / price` calculation will result in rounding down to 0. This means that the auction owner would still get the whole bid amount, but the bidder will get 0 `reserveToken`s to claim, resulting in a total loss of his bid.

The issue is also present when you are using same decimals tokens for both `reserve` and `purchase` tokens but the `volume` in a bid is less than the `price`. Again, the division will round down to zero, resulting in a 100% loss for the bidder.

Recommendations

In `commitBid` enforce that `volume >= price` and in `createAuction` enforce that the `reserveToken` decimals are equal to the `purchaseToken` decimals.

8.3. Medium Findings

[M-01] Missing input validation on `createAuction` function parameters can lead to loss of value

Severity

Impact: High, as it can lead to stuck funds

Likelihood: Low, as it requires user error/misconfiguration

Description

There are some problems with the input validation in `createAuction`, more specifically related to the timestamp values.

1. `endTimestamp` can be equal to `startTimestamp`, so `duration` will be 0
2. `endTimestamp` can be much further in the future than `startTimestamp`, so `duration` will be a huge number and the auction may never end
3. Both `startTimestamp` and `endTimestamp` can be much further in the future, so auction might never start

Those possibilities should all be mitigated, as they can lead to the initial reserves and/or the bids being stuck in the protocol forever.

Recommendations

Use a minimal `duration` value, for example 1 day, as well as a max value, for example 20 days. Make sure auction does not start more than X days after it has been created as well.

[M-02] Loss of precision in `scalarPrice` function

Severity

Impact: Medium, as the price will not be very far from the expected one

Likelihood: Medium, as it will not always result in big loss of precision

Description

In `scalarPrice` there is this code:

```
uint256 b_18 = 1e18;
uint256 t_mod = t % (t_r - t);
uint256 x = (t + t_mod) * b_18 / t_r;
uint256 y = !isInitialised ? state.price : window.price;

return y - (y * x) / b_18;
```

Here, when you calculate `x` you divide by `t_r` even though later you multiply `x` by `y`. To minimize loss of precision you should always do multiplications before divisions, since Solidity just rounds down when there is a remainder in the division operation.

Recommendations

Always do multiplications before divisions in Solidity, make sure to follow this throughout the whole `scalarPrice` method.

[M-03] Protocol won't work correctly with tokens that do not revert on failed `transfer`

Severity

Impact: High, as it can lead to a loss of value

Likelihood: Low, as such tokens are not so common

Description

Some tokens do not revert on failure in `transfer` or `transferFrom` but instead return `false` (example is ZRX). While such tokens are technically compliant with the standard it is a common issue to forget to check the return value of the `transfer`/`transferFrom` calls. With the current code, if such a call fails but

does not revert it will result in inaccurate calculations or funds stuck in the protocol.

Recommendations

Use OpenZeppelin's `SafeERC20` library and its `safe` methods for ERC20 transfers.

[M-04] Auction won't work correctly with fee-on-transfer & rebasing tokens

Severity

Impact: High, as it can lead to a loss of value

Likelihood: Low, as such tokens are not so common

Description

The code in `createAuction` does the following:

```
IERC20(reserveToken).transferFrom(msg.sender, address(this), reserveAmount);
...
...
state.reserves = reserveAmount;
```

so it basically caches the expected transferred amount. This will not work if the `reserveToken` has a fee-on-transfer mechanism, since the actual received amount will be less because of the fee. It is also a problem if the token used had a rebasing mechanism, as this can mean that the contract will hold less balance than what it cached in `state.reserves` for the auction, or it will hold more, which will be stuck in the protocol.

Recommendations

You can either explicitly document that you do not support tokens with a fee-on-transfer or rebasing mechanism or you can do the following:

1. For fee-on-transfer tokens, check the balance before and after the transfer and use the difference as the actual amount received.

2. For rebasing tokens, when they go down in value, you should have a method to update the cached `reserves` accordingly, based on the balance held. This is a complex solution.
3. For rebasing tokens, when they go up in value, you should add a method to actually transfer the excess tokens out of the protocol.

8.4. Low Findings

[L-01] Auction with `price == 0` can be re-created

The `createAuction` method checks if auction exists with this code

```
Auction storage state = _auctions[auctionId];

if (state.price != 0) {
    revert AuctionExists();
}
```

But the method does not check if the `startingOriginPrice` argument had a value of 0 - if it did, then `state.price` would be 0 in the next `createAuction` call. Even though this is not expected to happen, if it does it can lead to this line of code being executed twice:

```
IERC20(reserveToken).transferFrom(msg.sender, address(this), reserveAmount);
```

which will result in a loss for the caller. Make sure to require that the value of `startingOriginPrice` is not 0.

[L-02] The `commitBid` method does not follow Checks-Effects-Interactions pattern

It's a best practice to follow the CEI pattern in methods that do value transfers. Still, it is not always the best solution, as ERC777 tokens can still reenter while the contract is in a strange state even if you follow CEI. I would recommend adding a `nonReentrant` modifier to `commitBid` and also moving the `transferFrom` call to the end of the method.

[L-03] The `scalarPrice` method should have an `activeAuction` modifier

If an auction is inactive then the `scalarPrice` method will still be returning a price, even though it should not, since auction is over. Add the `activeAuction` modifier to it.

8.5. QA Findings

[QA-01] Using `require` statements without error strings

The `activeAuction` and `inactiveAuction` modifiers use `require` statements without error strings. Use `if` statements with custom errors instead for a better error case UX.

[QA-02] Typos in code

Fix all typos in the code:

`Ancoded` -> `Encoded`

`exipration` -> `expiration`

`fuflfillment` -> `fulfillment`

`operatorAdress` -> `operatorAddress`

`Uinx` -> `Unix`

`multipled` -> `multiplied`

[QA-03] Missing License identifier

The `RDA.sol` file is missing License Identifier as the first line of the file, which is a compiler warning. Add the `No License` license at least to remove the compiler warning.

[QA-04] Function state mutability can be restricted to view

The `scalarPriceUint` method does not mutate state but is not marked as `view` - add the `view` keyword to the function's signature.

[QA-05] Incomplete NatSpec docs

Methods have incomplete NatSpec docs, for example the `elapsedTime` method is missing the `@param timestamp` in its NatSpec, and also most methods are missing the `@return` param - for example `balancesOf` and `createAuction`. Make sure to write complete and detailed NatSpec docs for each public method.

[QA-06] Missing `override` keyword

The `createAuction`, `withdraw` and `redeem` methods are missing the `override` keyword even though the override methods from the `IRDA` interface. Add it to the mentioned methods.