Published in New Alchemy

New Alchemy Follow

Jun 21, 2018 · 10 min read · Listen

Save

AUX Platform Security Audit

Auctiond, Inc (AUX) engaged New Alchemy to audit the smart contracts for their Descending Price auctions.

The engagement was technical in nature and focused on identifying security flaws in the design and implementation of the contract, finding differences between the contract's implementation and their behaviour as described in public documentation, and finding any other issues with the contracts that may impact their trustworthiness. Auctiond, Inc provided New Alchemy with access to the relevant source code and whitepaper.

The audit was performed over 4 days. This document describes the issues discovered in the audit.

The code reviewed by New Alchemy is in the GitHub repository https://github.com/AuxPlatform/auction-contracts at commit hash `0da231969b7f791cb1c131956b3f8886f860617b` . The following files were audited:

- `DescendingPriceCryptoKitty.sol`
- `DescendingPriceERC20Auction.sol`
- `DescendingPriceERC721Auction.sol`

New Alchemy's audit was additionally guided by Auctiond, Inc's whitepaper and website.

Re-test notes: The revised contracts after the initial report was delivered are in commit `bbf266df2a85c8988df85950105d0905a8b6c989` .

The audit identified a variety of issues with impacts ranging from very minor to critical. The most significant finding involves concerns about the potential for miners to perform front-running attacks to ensure always winning auctions at the best price.

A portion of the code was standard and copied from widely-used and reviewed OpenZeppelin[1] contracts and, as a result, many of the findings pertained to the custom portions of code related to handling the auction itself.

Re-test notes: This re-test contains all the original findings from the previous version of the report; this document focuses on the comments and changes that Auctiond, Inc made after receiving the previous version of the report.

General Discussion

The contracts implement time-based descending price auctions. The contracts incorporates some code from the open-source OpenZeppelin project.

The code in 'DescendingPriceCryptoKitty.sol' specifies Solidity version 0.4.11, which is older than the the recommended latest version at the time of the audit, which is

Solidity. The code in the two other files specify Solidity version 0.4.23 which is only slightly behind the current recommended version. Auctiond, Inc should use the latest version of Solidity in order to take advantage of the latest features and improvements that Solidity has to offer.

The code implements and uses the SafeMath library, which defines functions for safe math operations that will throw errors in the cases of integer overflow or underflows. SafeMath is used in a consistent and safe manner throughout the audited contract which is a good practice and effectively protects the contract from overflows.

The code also relies on block numbers instead of system time at every place where there is a reference to time or duration. This is a good practice and effectively prevents time manipulation by miners.

One critical issue was found where miners could abuse their position to attempt a front-running type attack to win auctions before legitimate bidders. This issue breaks the basic assumptions that in a descending auction, the first bidder wins the auction and that no bidder knows when a bid is going to be made by another bidder (potentially driving the price higher for the benefit of the seller).

Re-test notes: Auctiond, Inc elected not to specify the version of Solidity used in the contracts at this time and have plans to upgrade to the most recent version of Solidity in the near future. Auctiond, Inc fixed some of the issues that were reported in the original report, and details about these fixes or the response provided by Auctiond, Inc to the findings can be found in each finding description.

Critical Issues

Not Fixed: Miners could take advantage of their position to win auctions

This issue affects all three of the `DescendingPriceAuction` contracts defined in the audited files. A miner seeing a transaction to bid on an auction could take advantage of knowing of the transaction before it has actually been mined to issue a similar transaction with a higher gas price in an attempt to win the bid before the legitimate

 

- This issue breaks the basic assumption that in a descending auction, the first bidder wins the auction.
- This issue breaks the basic assumption that no bidder knows when a bid is going to be made by another bidder, which usually drives the price higher for the benefit of the seller.

While the argument could be made that the seller is not at a disadvantage since the sale has been made at a valid price, the higher gas price put on the transaction by the miner rewards another miner (or itself) while that value should have more rightfully have gone into rewarding the seller for the higher demand of the auction's reward.

This issue is often referred to as front-running and is described in depth as well as some of its potential solutions in the following papers:

- https://blog.0xproject.com/front-running-griefing-and-the-perils-of-virtual-settlement-part-1-8554ab283e97
- https://hackernoon.com/front-running-bancor-in-150-lines-of-python-with-ethereum-api-d5e2bfd0d798
- http://hackingdistributed.com/2017/08/28/submarine-sends/

New Alchemy recommends overhauling the bidding system and bidding resolution to prevent such issues in descending price auctions.

Auctiond, Inc provided the following response to the issue described above:

"Frontrunning is a known problem for the Ethereum network (or any proof-of-work distributed ledger technology with public transaction data). While AUX agrees that the frontrunning vulnerability outlined in this audit breaks the assumption that in a descending price auction, bidders' intentions and valuations are completely private until the auction is settled, we believe this does not represent a critical business issue for the following reasons:

1. Any frontrunning bidder would, by definition, be willing to pay the current price (plus either a higher gas fee, or control sufficient hashpower to effectively

This means that any frontrunning bidder would have to have private valuations materially similar to the next highest bidder, as well as an interest in the auctioned asset and the means to frontrun the auction. Moreover, their behavior would not necessarily be distinguishable from a 'lucky' bidder who increased their gas amount in a blind attempt to win.

Ultimately, AUX believes that the possibility of frontrunning in descending price auctions is currently an acceptable tradeoff in order to ensure a good user experience, since all known mitigation techniques substantially increase user friction. As AUX moves toward the launch of the full peer-to-peer network, we will reevaluate this tradeoff and implement smart contracts with stricter settlement processes to better mitigate these types of attacks as needed.

Re-test notes: Auctiond, Inc has elected not to fix this issue at this time. New Alchemy stands by its original assessment that this issue is critical. It provides an unfair advantage to anyone willing to exploit this issue to win auctions at the best price without having to make blind bids like any other legitimate user would, and that it clearly disadvantages the seller that cannot count on the complete secrecy of bids to drive the bidding price higher.

Minor Issues

Fixed: Limitations on integer math could prevent price changes

The process of calculating the auction's current price at a specific block (or point in time) is defined in the function `getCurrentPrice` (line 369 in `DescendingPriceCryptoKitty.sol` and line 270 in both `DescendingPriceERC20Auction.sol` and `DescendingPriceERC721Auction.sol`). This calculation involves a division between `priceDifference` and `blockDifference`. While all mathematical operations in those functions use safeMath to securely prevent any potential overflow, the result of that division could be 0, which in turn means that the price would stay the same for the whole duration of the auction (preventing the price to descend progressively towards the final price set by the auction owner).

For this specific issue to happen, the priceDifference would just need to be lower than

```
function createAuction(uint256 startPrice, uint256
priceFloor,uint256 duration, ... {
    require(startPrice > 0 && priceFloor < startPrice && priceFloor
>= 0 && duration > 0);
    require((startPrice - priceFloor) >= (duration))
```

Re-test notes: Auctiond, Inc improved the precision of integer math operations by reversing the order of division and multiplication. Using integer-only division with discarded remainders will always result in some 'descending price lumpiness'. However, the very large magnitude of the price differences in the numerator relative to the comparatively small magnitude of block differences in the denominator means this lumpiness is essentially insignificant noise in the larger business context.

Not Fixed: Lack of two-phase ownership transfer

In contracts that inherit the common `Ownable` contract from the OpenZeppelin project^4, a contract has a single owner. That owner can unilaterally transfer ownership to a different address. However, if the owner of a contract makes a mistake in entering the address of an intended new owner, then the contract can become irrecoverably unowned.

In order to preclude this, New Alchemy recommends implementing two-phase ownership transfer. In this model, the original owner designates a new owner, but does not actually transfer ownership. The new owner then accepts ownership and completes the transfer. This can be implemented as follows:

```
contract Ownable {
    address public owner;
    address public newOwner

    event OwnershipTransferred(address indexed oldOwner, address
indexed newOwner);

    function Ownable() public {
        owner = msg.sender;
        newOwner = address(0);
    }
```

 

```
        require(address(0) != _newOwner);
        newOwner = _newOwner;
    }

    function acceptOwnership() public {
        require(msg.sender == newOwner);
        OwnershipTransferred(owner, msg.sender);
        owner = msg.sender;
        newOwner = address(0);
    }
}
```

Re-test notes: Auctiond, Inc elected not to fix this issue at this time and to instead keep in line with the contracts as they are provided by OpenZeppelin.

Not Fixed: Function Checking Asset Ownership Does Not Check on the External Contract

For the `CryptoKittyAuction`, `ERC721Auction`, and `ERC20Auction` contracts, the `auctionHasKitty`, `auctionHasAsset`, and `auctionHasAssets` functions do not actually check if the contract owns the assets at the moment it is called (only that the auction was correctly created and not ended).

```
function auctionHasKitty(uint256 auctionId) private view returns
(bool) {
    uint256 kittyId = auctionIdToKittyId[auctionId];

    //An auctionId of 0 represents a non-existent auction, which
means the kitty isn't in any auction managed by this contract.
    uint256 auctionThatCurrentlyOwnsKitty =
kittyIdToAuctionId[kittyId];

    return(auctionThatCurrentlyOwnsKitty == auctionId);
  }

function auctionHasAssets(uint256 auctionId) private view returns
(bool) {
    return (auctionIdToAmount[auctionId] != 0);
  }

function auctionHasAsset(uint256 auctionId) private view returns
(bool) {
    address assetContractForAuction =
auctionIdToAssetContract[auctionId];
```

```
    uint256 auctionThatCurrentlyOwnsAsset =
assetContractToAssetIdToAuctionId[assetContractForAuction][assetId];

    return(auctionThatCurrentlyOwnsAsset == auctionId &&
auctionThatCurrentlyOwnsAsset != 0);
  }
```

It would be preferable that the `auctionHasAssets` function from `ERC20Auction` checks the contract's balance for the ERC20 token (using `balanceOf`) to ensure that the tokens were not moved prior to the bid. It would also be preferable that the `auctionHasAsset` function from `ERC721Auction` checks the contract's owns the assetId being sold (using `ownerOf`) to ensure that property of the asset was not changed prior to the bid. Similarly, it would also be preferable that the `auctionHasKitty` function from `CryptoKittyAuction` checks the contract's owns the assetId being sold (using `ownerOf`) to ensure that property of the asset was not changed prior to the bid.

While the risk of property of the tokens or asset having changed between creation of the auction and the moment the bid is made is slim and the transfer should in theory fail if it can not be performed (reverting the bid when it does), adding a check at the moment of the bid would help improve the user's trust in that the contract can indeed transfer the auctioned item even in case of a contract not entirely respecting the CryptoKitty, ERC20, or ERC721 standards. Investigating the contracts being auctioned on to ensure they do correctly implement the CryptoKitty, ERC20, or ERC721 standards would also mitigate this risk.

Re-test notes: Auctiond, Inc elected not to fix this issue at this time.

Line by line comments

This section lists comments on design decisions and code quality made by New Alchemy during the review. They are not known to represent security flaws.

DescendingPriceCryptoKitty.sol


surplus to the message sender).

Re-test notes: Auctiond, Inc effectively fixed this issue by changing the comment above the call to the `requiresFee` modifier.

Fixed: Line 344: same modifier used twice

The `bid` function from `DescendingPriceAuction` has the `whenNotPaused` modifier declared twice in its function prototype. Only setting the modifier once would make the code easier to read.

Re-test notes: Auctiond, Inc effectively fixed this issue by removing the second `whenNotPaused` modifier.

DescendingPriceERC721Auction.sol

Fixed: Line 443: inaccurate comment

The comment references `cat` which is probably a copy-paste remnant from the `DescendingPriceCryptoKitty.sol` file but does not reflect the actual code being commented.

Re-test notes: Auctiond, Inc effectively fixed this issue by changing the comment to more accurately describe the function underneath.

descendingpriceERC20Auction.sol

Fixed: Line 245: same modifier used twice

The `bid` function from `DescendingPriceAuction` has the `whenNotPaused` modifier declared twice in its function prototype. Only setting the modifier once would make the code easier to read.

Re-test notes: Auctiond, Inc effectively fixed this issue by removing the second `whenNotPaused` modifier.

any other statements about fitness of the contracts to purpose, or their bug-free status. The audit documentation is for discussion purposes only.

New Alchemy *is a leading blockchain strategy and technology group specializing in ICO services, security audits, and revenue/capital solutions for the most innovative companies and tokenization projects worldwide.* ***Get in touch with us at Hello@NewAlchemy.io***