Starknet Auction Audit Report

Version 1.0

0xBlockPay

November 14, 2024

Starknet Auction Audit Report

0xBlockPay

October 10, 2024

Prepared by: 0xBlockPay

Lead Security Researcher: - eth0x

Table of Contents

Protocol Summary

The Starknet Auction protocol is a simple auction protocol that enables the auction of NFTs using ERC20 tokens as bid currency. The NFT owner is the admin of the protocol. The NFT owner deploys the protocol and starts the auction. The participants in the auction (called users/bidders) place bids. Each bid should be bigger than the previous highest bid. The participants may have multiple bids. After the auction time expires the bidder who made the highest bid wins the auction and receives the NFT. The owner of the NFT receives the value of the highest bid and the other participants can withdraw their unsuccessful bids.

Disclaimer

The 0xBlockPay team makes all effort to find as many vulnerabilities in the code in the given time period, but holds no responsibilities for the findings provided in this document. A security audit by the team is not an endorsement of the underlying business or product. The audit was time-boxed and the review of the code was solely on the security aspects of the Solidity implementation of the contracts.

Risk Classification

		Impact		
		High	Medium	Low
	High	H	H/M	M
Likelihood	Medium	H/M	M	M/L
	Low	M	M/L	L

We use the CodeHawks severity matrix to determine severity. See the documentation for more details.

Audit Details

Scope

The findings describe in this document correspond the following repository and commit hash:

```
git clone https://github.com/Cyfrin/2024-10-starknet-auction.git
```

```
Commit Hash:
```

In Scope:

```
src/
#-- starknet_auction.cairo
```

- Blockchains: Starknet
- Tokens: STRK

Roles

- NFT Owner/Admin (Trusted) - Initiates the auction and receives the value of the highest bid.
- Bidders/Users - Place bids and can withdraw their bids if they don't win. The bidder with the highest bid wins the NFT.

Executive Summary

*We spend 10 hours with 1 auditor using foundry, aderyn and slither tools.

Issues found

Severity	Number of issues found
High	5
Medium	1
Low	1
Info	0
Gas	0
Total	7

Findings

High

[H-1] Bidders bids except last are lost

Summary

If a bidder submits multiple bids the Map data structure will preserve his last bid against his address and all the earlier bids could not be tracked and hence when the auction is over his bids can not be withdrawn as they are not saved inside the contract.

Vulnerability Details

As you can see from the following test, the user bids 3 times but only his last bid is stored inside the map and his earlier bids are not saved.

```
#[test]
fn test_bids_lost() {
    let (auction_dispatcher, auction_contract, erc20_contract_address,
        _) =
        deploy_auction_contract();
    //The owner calls the start function and the auction begins.
    auction_dispatcher.start(86400, 10);

    let erc20_dispatcher = IMockERC20TokenDispatcher { contract_address
        : erc20_contract_address };
    //Change the caller address
    let first_bidder_address: ContractAddress = 123.try_into().unwrap()
        ;
    start_cheat_caller_address_global(first_bidder_address);

    erc20_dispatcher.mint(first_bidder_address, 60);
    // let balance = erc20_dispatcher.token_balance_of(
        first_bidder_address);
    // assert(balance == 20, 'Balance must be 20');

    erc20_dispatcher.token_approve(auction_contract, 60);

    //The first bidder calls the bid function with amount of 11.
    auction_dispatcher.bid(15);

    let values = load(
        auction_contract,
        map_entry_address(
            selector!("bid_values"), array![first_bidder_address.try_into().unwrap()].span(),
        ),
        3,
    );
    println!("Values for first_bidder {:?}", values.span());

    auction_dispatcher.bid(20);

    let values = load(
        auction_contract,
        map_entry_address(
            selector!("bid_values"), array![first_bidder_address.try_into().unwrap()].span(),
        ),
        3,
    );
    println!("Values for first_bidder {:?}", values.span());

    auction_dispatcher.bid(25);

    stop_cheat_caller_address_global();

    let values = load(
        auction_contract,
        map_entry_address(
            selector!("bid_values"), array![first_bidder_address.try_into().unwrap()].span(),
        ),
        3,
    );
    println!("Values for first_bidder {:?}", values.span());
}
```

Output

```
Values for first_bidder [15, 0, 0]
Values for first_bidder [20, 0, 0]
Values for first_bidder [25, 0, 0]
[PASS] starknet_auction_integrationtest::test_contract::test_bids_lost (gas: ~1996)
Running 0 test(s) from src/
```

Impact

The bidder will not be able to withdraw his bids other than the last one when the auction is ended.

Tools Used

Tests

Recommendations

Instead of saving current bid against sender address in map, sum of current bid and previous bid should be saved inside map against sender. This way we can keep track of total amount that someone has bid and when the auction is ended we can return this amount for all the bidders and for the winner we can subtract highest bidding from this amount and return the remaining amount to winner in case if he has bidded multiple times.

Other functions in the contract need to be changed as well.

```
        fn bid(ref self: ContractState, amount: u64) {
            let time = get_block_timestamp();
            let erc20_dispatcher = IERC20Dispatcher { contract_address: self.erc20_token.read() };
            let sender = get_caller_address();
            let receiver = get_contract_address();
            let current_bid = self.highest_bid.read();

            assert(self.started.read(), 'Auction is not started');
            assert(time < self.bidding_end.read(), 'Auction ended');
            assert(amount > current_bid, 'The bid is not sufficient');
+            let old_bid = self.bid_values.read(sender);
+            self.bid_values.entry(sender).write(amount + old_bid);
-            self.bid_values.entry(sender).write(amount);
            self.emit(NewHighestBid {amount: self.highest_bid.read(), sender: sender});
            self.highest_bidder.write(sender);
            self.highest_bid.write(amount);

            erc20_dispatcher.transfer(receiver, amount.into());
        }
```

```
Output of same test with obove modification.
Values for first_bidder [15, 0, 0]
Values for first_bidder [35, 0, 0]
Values for first_bidder [60, 0, 0]
[PASS] starknet_auction_integrationtest::test_contract::test_bids_lost (gas: ~2000)
```

[H-2] Highest bidder can still withdraw and get back the bid token on top of owning the NFT

Summary

Highest bidder can still get back their bid token by making the call to `withdraw` function, allowing them to get and own the NFT for free

https://github.com/SiewLin-AIDeaBuddy/first-flight-2024-10-starknet-auction/blob/04f8a8b7f16edc7aeba06a3a6195e0257ddbefd0/src/starknet_auction.cairo#L131-L134

Vulnerability Details

The auction protocol will transfer the NFT to highest bidder when owner ends the auction. Owner shall then be able to withdraw the highest bid amount whilst other bidders could withdraw their unsuccessful bid tokens back. However, highest biddest could still call the `withdraw` function, allowing them to get back their bid token amount and still owns the NFT towards the ends. As the highest bidder makes the withdrawal which they are not supposed to, it causes other users including the owner who is eligible to make the withdraw could fail to withdraw their tokens back due to insufficient balance in the protocol.

Proof of concept:

In `test\test_contract.cairo`, add the following test case:

```
#[test]
#[should_panic(expected: 'ERC20: insufficient balance')]
fn test_audit_highest_bidder_can_still_withdraw() {
    let (auction_dispatcher, auction_contract, erc20_contract_address,
        erc721_contract_address) = deploy_auction_contract();
    let erc20_dispatcher = IMockERC20TokenDispatcher { contract_address
        : erc20_contract_address };
    let erc721_dispatcher = IERC721Dispatcher { contract_address:
        erc721_contract_address };

    //The owner calls the start function and the auction begins.
    auction_dispatcher.start(86400, 10);

    // first bidder action
    let first_bidder_address: ContractAddress = 123.try_into().unwrap()
        ;
    start_cheat_caller_address_global(first_bidder_address);

    erc20_dispatcher.mint(first_bidder_address, 15);
    erc20_dispatcher.token_approve(auction_contract, 15);

    auction_dispatcher.bid(15); // calls the bid function with amount of 15.
    stop_cheat_caller_address_global();

    // second bidder action
    let second_bidder_address: ContractAddress = 456.try_into().unwrap();
    start_cheat_caller_address_global(second_bidder_address);

    erc20_dispatcher.mint(second_bidder_address, 18);
    erc20_dispatcher.token_approve(auction_contract, 18);

    auction_dispatcher.bid(18); // calls the bid function with amount of 18.

    // Check the token balance after all biddings
    let erc20_balance_after_all_bids = erc20_dispatcher.token_balance_of(auction_contract);
    println!("erc20_balance_after_all_bids: {}", erc20_balance_after_all_bids);

    // Check the token balance of the highest bidder (second-bidder) after bid
    let erc20_balance_second_bidder_after_bid = erc20_dispatcher.token_balance_of(second_bidder_address);
    assert(erc20_balance_second_bidder_after_bid == 0, 'Balance must be 0');
    println!("erc20_balance_second_bidder_after_bid: {}", erc20_balance_second_bidder_after_bid);

    // Check the nft balance of the highest bidder (second-bidder) after bid
    let nft_balance_second_bidder_after_bid = erc721_dispatcher.balance_of(second_bidder_address);
    assert(nft_balance_second_bidder_after_bid == 0, 'NFT balance must be 0');
    println!("nft_balance_second_bidder_after_bid: {}", nft_balance_second_bidder_after_bid);

    stop_cheat_caller_address_global();

    let time = get_block_timestamp();
    start_cheat_block_timestamp(auction_contract, time + 86401);

    start_cheat_caller_address_global(auction_contract);
    erc20_dispatcher.token_approve(auction_contract, 18);
    erc20_dispatcher.token_approve(first_bidder_address, 15);
    erc20_dispatcher.token_approve(second_bidder_address, 18);
    erc721_dispatcher.approve(second_bidder_address, 1);
    stop_cheat_caller_address_global();

    // auction period ends
    auction_dispatcher.end();

    // highest bidder (second-bidder) makes withdrawal despite already owning the NFT
    start_cheat_caller_address_global(second_bidder_address);
    auction_dispatcher.withdraw();
    stop_cheat_caller_address_global();

    // Check the token balance of the highest bidder (second-bidder) after withdraw
    let erc20_balance_second_bidder_after_withdraw = erc20_dispatcher.token_balance_of(second_bidder_address);
    assert(erc20_balance_second_bidder_after_withdraw == 18, 'Balance must be 18');
    println!("erc20_balance_second_bidder_after_withdraw: {}", erc20_balance_second_bidder_after_withdraw);

    // Check the nft balance of the highest bidder after bid and after withdraw
    let nft_balance_second_bidder_after_withdraw = erc721_dispatcher.balance_of(second_bidder_address);
    assert(nft_balance_second_bidder_after_withdraw == 1, 'NFT balance must be 1');
    println!("nft_balance_second_bidder_after_withdraw: {}", nft_balance_second_bidder_after_withdraw);

    // owner's action to withdraw the highest bid token amount exchanging NFT for the token
    auction_dispatcher.withdraw(); // expect to panic for `ERC20: insufficient balance`

    stop_cheat_block_timestamp(auction_contract);
}
```

Run the test `snforge test --features enable_for_tests test_audit_highest_bidder_can_s`

```
...

Collected 1 test(s) from starknet_auction package
Running 1 test(s) from tests/
erc20_balance_after_all_bids: 33
erc20_balance_second_bidder_after_bid: 0
nft_balance_second_bidder_after_bid: 0
erc20_balance_second_bidder_after_withdraw: 18
nft_balance_second_bidder_after_withdraw: 1
[PASS] starknet_auction_integrationtest::test_contract::test_audit_highest_bidder_can_still_withdraw (gas: ~2319)
```

The test passes, indicating that highest bidder is able to withdraw the bid amount of 18 and yet still owns the NFT. Whilst the owner who only makes the `withdraw` call after the highest bidder will fail to withdraw due to insufficient balance in the protocol. The test case supported the panic via `#[should_panic(expected: 'ERC20: insufficient balance')]`

Impact

Highest bidder could still withdraw the bid token despite already successfully bidding and owning the NFT whilst users who are eligible to withdraw their bid tokens back could fail to withdraw due to insufficient balance left in the protocol

Tools Used

Manual review with test

Recommendations

Make amendment to the `withdraw` function to disable highest bidder make withdrawal

```
fn withdraw(ref self: ContractState) {
    assert(self.started.read(), 'Auction is not started');
    assert(self.ended.read(), 'Auction is not ended');

    let caller = get_caller_address();
    let sender = get_contract_address();
    let erc20_dispatcher = IERC20Dispatcher { contract_address: self.
        erc20_token.read() };
    let amount = self.bid_values.entry(caller).read();
    let amount_owner = self.highest_bid.read();

    if caller == self.nft_owner.read() {
        self.highest_bid.write(0);
        erc20_dispatcher.transfer_from(sender, caller, amount_owner.
            into());
    }

    if amount > 0 {
+       assert(caller != self.highest_bidder.read(), 'Not eligible to
withdraw');
        let sender = get_contract_address();
        erc20_dispatcher.transfer_from(sender, caller, amount.into());
    }

    self.emit(Withdraw {amount: amount, caller: caller});
}
```

[H-3] Multiple Withdrawal Vulnerabilities

Summary

bidders can withdraw multiple times due to lack of checks in the `withdraw` function.

Vulnerability Details

https://github.com/Cyfrin/2024-10-starknet-auction/blob/main/src/starknet_auction.cairo#L116-L137

- Multiple Withdrawals for All Bidders: The `withdraw` function allows all bidders, including non-winners, to withdraw their bids multiple times. This is because the bid amounts are never reset after withdrawal.
- Auction Winner Can Reclaim Funds: The contract doesn't distinguish between the winning bidder and other bidders in the withdrawal process. This allows the auction winner to withdraw their bid, effectively undoing the auction result.
- Lack of State Tracking: The contract doesn't track whether a withdrawal has occurred or the auction has been settled, allowing for these incorrect behaviors.

POC

```
#[test]
fn test_multiple_vulnerabilities_in_withdraw() {
    let (auction_dispatcher, auction_contract, erc20_contract_address,
        _) = deploy_auction_contract();
    auction_dispatcher.start(86400, 10);

    let erc20_dispatcher = IMockERC20TokenDispatcher { contract_address
        : erc20_contract_address };

    // First bidder
    let bidder1: ContractAddress = 123.try_into().unwrap();
    start_cheat_caller_address_global(bidder1);
    erc20_dispatcher.mint(bidder1, 20);
    erc20_dispatcher.token_approve(auction_contract, 20);
    auction_dispatcher.bid(15);
    stop_cheat_caller_address_global();

    // Second bidder (winner)
    let bidder2: ContractAddress = 124.try_into().unwrap();
    start_cheat_caller_address_global(bidder2);
    erc20_dispatcher.mint(bidder2, 25);
    erc20_dispatcher.token_approve(auction_contract, 25);
    auction_dispatcher.bid(20);
    stop_cheat_caller_address_global();

    // End the auction
    let time = get_block_timestamp();
    start_cheat_block_timestamp(auction_contract, time + 86401);
    auction_dispatcher.end();

    // Test multiple withdrawals by non-winning bidder
    start_cheat_caller_address_global(bidder1);
    let initial_balance1 = erc20_dispatcher.token_balance_of(bidder1);
    auction_dispatcher.withdraw();
    let balance_after_first_withdrawal1 = erc20_dispatcher.
        token_balance_of(bidder1);
    auction_dispatcher.withdraw();
    let balance_after_second_withdrawal1 = erc20_dispatcher.
        token_balance_of(bidder1);
    assert(balance_after_second_withdrawal1 >
        balance_after_first_withdrawal1, 'Non-winner should not withdraw
         twice');
    stop_cheat_caller_address_global();

    // Test withdrawal by winning bidder
    start_cheat_caller_address_global(bidder2);
    let initial_balance2 = erc20_dispatcher.token_balance_of(bidder2);
    auction_dispatcher.withdraw();
    let balance_after_withdrawal2 = erc20_dispatcher.token_balance_of(
        bidder2);
    assert(balance_after_withdrawal2 > initial_balance2, 'Winner should
         not be able to withdraw');
    stop_cheat_caller_address_global();
}
```

Impact

- Financial Loss: The contract could be drained of more funds than were actually bid, leading to significant financial loss.
- Auction Integrity: The ability for the winner to withdraw their bid undermines the entire auction process, as the highest bidder can effectively cancel their win.
- Unfair Advantage: Bidders who realize this vulnerability could exploit it to withdraw more than

their initial bid, potentially profiting unfairly.

Tools Used

manual review

Recommendations

- Implement a withdrawal tracking mechanism:

```
    self.bid_values.entry(caller).write(0);
```

- Distinguish between the winner and other bidders:

```
if caller == self.highest_bidder.read() {
  assert(false, 'Winner cannot withdraw');
}
```

Add this check to prevent the auction winner from withdrawing their bid.

- Implement an auction settlement process: After the auction ends, transfer the winning bid to the NFT owner and the NFT to the winner automatically, rather than relying on manual withdrawals.

[H-4] Incorrect Use Of transfer() In bid()

Summary

`transfer()` is incorrectly being used within `bid()` to deposit tokens from bidders.

Vulnerability Details

On line 113 `transfer()` is being used to deposit ERC20 tokens from bidders into the contract. This means that the sender is actually the smart contract and not the bidder who is trying to deposit tokens.

Impact

If the contract has no preexisting balance this will result in the call failing with an `INSUFFICIENT_BALANCE` error. This means that all calls to `bid()` would fail. However, if the contract had a preexisting balance, it means that bidders would be able to bid using the contracts balance and not use their own tokens. This would essentially allow them to withdraw this amount from the contract upon completion of the auction. Also, it would allow the winning bidder to win the NFT and have the contract pay for it.

Tools Used

Manual Review

Recommendations

Update line 113 to the following: `erc20_dispatcher.transferFrom(sender, receiver, amount.into());`

[H-5] ERC20 tokens will cause an overflow

Summary

ERC20 tokens used in the auction that are more than `uint64.max()` will cause an overflow

Vulnerability Details

uint64 is so small amount to express an ERC20 token, we will not be able to actually use most of normal transfer functions.

type(uint64).max = 18_446_744_073_709_551_615 ~= 18.45e18.

Since the common decimal is 18, this means that if the amount of tokens to transfer/bid exceeds 18.45 we will not be able to recover them, as this will result in overflow.

If we say that the token amount worth 1$, this means that we will not be able to do the transfer that's worth 19$ or more.

Impact

this will result in the inability to use most of ERC20 transfers/bids because of extremely low data size for assets.

Tools Used

Manual review

Recommendations

Change the type of tokens to uint256 instead of uint64.

Medium

[M-1] Owner is unable to get back the NFT if no bids were placed by the end of the auction period

Summary

The NFT owner cannot retrieve back the NFT ownership if no bids are placed by the end of the auction period. The owner will be unable to execute the `end` function as it will revert with a `No bids` error message, leaving the NFT gets stuck in the contract permanently.

https://github.com/SiewLin-AIDeaBuddy/first-flight-2024-10-starknet-auction/blob/04f8a8b7f16edc7aeba06a3a6195e0257ddbefd0/src/starknet_auction.cairo#L149

Vulnerability Details

When the auction period ends, owner can call `end` function to transfer the NFT to the highest bidder and receive the tokens bid amount from the highest bidder via `withdraw` function. However, if there's no bids were placed by the end of the auction period, owner won't be able to retrieve back the NFT from the contract due to the assertion check `assert(self.starting_price.read()< self.highest_bid.read(), 'No bids');` in the `end` function.

```
        fn end(ref self: ContractState) {
            let time = get_block_timestamp();
            let caller = get_caller_address();
            let erc721_dispatcher = IERC721Dispatcher {
                contract_address: self.erc721_token.read() };
            let sender = get_contract_address();

            assert(caller == self.nft_owner.read(), 'Not the nft owner'
                );
            assert(self.started.read(), 'Auction is not started');
            assert(time >= self.bidding_end.read(), 'Auction is not yet
                 ended');
            assert(!self.ended.read(), 'Auction end is already called');
<@@>!       assert(self.starting_price.read() < self.highest_bid.read(), 'No bids');

            self.ended.write(true);
            self.emit(End {highest_bid: self.highest_bid.read(), highest_bidder: self.highest_bidder.read()});

            erc721_dispatcher.transfer_from(sender, self.highest_bidder.read(), self.nft_id.read().into());
        }
```

When no bids were placed, the `self.starting_price` will be the same as `self.highest_bid`, and therefore causes the assertion at `assert(self.starting_price.read()< self.highest_bid.read(), 'No bids');` to fail with `No bids` error. As the owner can't complete the `end` function and there's no other way for the owner to retrive back the NFT when no bids were placed, owner's NFT will therefore get stuck in the contract permanently.

Proof of Concept:

Step 1: In `test\test_contract.cairo`, add the following test case:

```
#[test]
#[should_panic(expected: 'No bids')]
fn test_audit_owner_cant_get_back_nft() {
    let (auction_dispatcher, auction_contract, _, erc721_contract_address) = deploy_auction_contract();
    let erc721_dispatcher = IERC721Dispatcher { contract_address: erc721_contract_address };
    let time = get_block_timestamp();

    // The owner calls start function.
    auction_dispatcher.start(86400, 10);

    start_cheat_block_timestamp(auction_contract, time + 86401);
    auction_dispatcher.end();
    stop_cheat_block_timestamp(auction_contract);
}
```

Step 2: Run `snforge test --features enable_for_tests test_audit_owner_cant_get_back_`

```
...

Collected 1 test(s) from starknet_auction package
Running 1 test(s) from tests/
[PASS] starknet_auction_integrationtest::test_contract::test_audit_owner_cant_get_back_nft (gas: ~1680)
Running 0 test(s) from src/
```

```
Tests: 1 passed, 0 failed, 0 skipped, 0 ignored, 17 filtered out
```

The test passes with expected panic of `No bids` via `#[should_panic(expected: 'No bids')]`

Impact

Owner can't retrive back the NFT when no bids are placed by the end of the auction period.

Tools Used

Manual review

Recommendations

To make amendment on the assertion check as follow so owner will still be able to get back the NFT in the event that no bids are placed by end of auction period

```
fn end(ref self: ContractState) {
    let time = get_block_timestamp();
    let caller = get_caller_address();
    let erc721_dispatcher = IERC721Dispatcher { contract_address: self.
        erc721_token.read() };
    let sender = get_contract_address();

    assert(caller == self.nft_owner.read(), 'Not the nft owner');
    assert(self.started.read(), 'Auction is not started');
    assert(time >= self.bidding_end.read(), 'Auction is not yet ended');
    assert(!self.ended.read(), 'Auction end is already called');
-   assert(self.starting_price.read() < self.highest_bid.read(), 'No
    bids');

+    if (self.starting_price.read() < self.highest_bid.read()) {
+        assert(self.starting_price.read() < self.highest_bid.read(), '
    No bids');
+    }

    self.ended.write(true);
    self.emit(End {highest_bid: self.highest_bid.read(), highest_bidder:
        self.highest_bidder.read()});

    erc721_dispatcher.transfer_from(sender, self.highest_bidder.read(),
        self.nft_id.read().into());
}
```

Modify earlier test case to verify that the contract doesn't hold the NFT with the amendment suggested above when owner calls the `end` function even when there's no bids were placed by the end of the auction period.

```
#[test]
fn test_audit_owner_cant_get_back_nft() {
    let (auction_dispatcher, auction_contract, _,
        erc721_contract_address) = deploy_auction_contract();
    let erc721_dispatcher = IERC721Dispatcher { contract_address:
        erc721_contract_address };
    let time = get_block_timestamp();

    // The owner calls start function.
    auction_dispatcher.start(86400, 10);

    start_cheat_block_timestamp(auction_contract, time + 86401);

    start_cheat_caller_address_global(auction_contract);
    erc721_dispatcher.approve(auction_contract, 1);
    stop_cheat_caller_address_global();

    // before owner calls `end` function
    let nft_balance_before_end = erc721_dispatcher.balance_of(
        auction_contract);
    assert(nft_balance_before_end == 1, 'Nft balance must be 1');

    auction_dispatcher.end();

    // after owner calls `end` function
    let nft_balance_after_end = erc721_dispatcher.balance_of(
        auction_contract);
    assert(nft_balance_after_end == 0, 'Nft balance must be 0');

    stop_cheat_block_timestamp(auction_contract);
}
```

Run the test `snforge test --features enable_for_tests test_audit_owner_cant_get_back_` and it will pass

```
....

Collected 1 test(s) from starknet_auction package
Running 0 test(s) from src/
Running 1 test(s) from tests/
[PASS] starknet_auction_integrationtest::test_contract::
    test_audit_owner_cant_get_back_nft (gas: ~1769)
Tests: 1 passed, 0 failed, 0 skipped, 0 ignored, 17 filtered out
```

Low

[L-1] Incorrect event bid amount emitted in `StarknetAuction::bid` function creating confusion and impact downstream/front-end process that depends on this event for rendering information related to bid transaction

Summary

The event in `StarknetAuction::bid` function was found emitted before the `highest_bid` was updated with latest user bid amount. This causes the bid amount emitted by the event is still the previous bid amount and not the user bid amount, resulting incorrect information in the emitted event upon the successful of bid transaction.

https://github.com/SiewLin-AIDeaBuddy/first-flight-2024-10-starknet-auction/blob/04f8a8b7f16ed c7aeba06a3a6195e0257ddbefd0/src/starknet_auction.cairo#L109

Vulnerability Details

In `StarknetAuction::bid` function, the event was found emitted before the `highest_bid` was updated with latest user bid amount.

```
fn bid(ref self: ContractState, amount: u64) {
        let time = get_block_timestamp();
        let erc20_dispatcher = IERC20Dispatcher { contract_address:
            self.erc20_token.read() };
        let sender = get_caller_address();
        let receiver = get_contract_address();
        let current_bid = self.highest_bid.read();

        assert(self.started.read(), 'Auction is not started');
        assert(time < self.bidding_end.read(), 'Auction ended');
        assert(amount > current_bid, 'The bid is not sufficient');

        self.bid_values.entry(sender).write(amount);
        self.emit(NewHighestBid {amount: self.highest_bid.read(),
            sender: sender});
        self.highest_bidder.write(sender);
        self.highest_bid.write(amount);

        erc20_dispatcher.transfer(receiver, amount.into());
}
```

At the end of a successful bid process, the event will emit the previous bid amount and not the latest user bid amount. This will create confusion and impact any downstream/front-end process that relies on the emitted event for proper rendering of the bid transaction information.

Proof of Concept:

Step 1: Update the visibility of the following event and struct in `src\starknet_auction.cairo`

```
    #[event]
    #[derive(Drop, starknet::Event)]
-   enum Event {
+   pub enum Event {
        Started: Started,
        NewHighestBid: NewHighestBid,
        Withdraw: Withdraw,
        End: End,
    }

    #[derive(Drop, starknet::Event)]
-   struct NewHighestBid {
-        amount: u64,
-        sender: ContractAddress,
+   pub struct NewHighestBid {
+        pub amount: u64,
+        pub sender: ContractAddress,
    }
```

Step 2: In `tests\test_contract.cairo`, add the following import and test

```
// add these imports at the top
use snforge_std::{spy_events, EventSpyAssertionsTrait};
use starknet_auction::starknet_auction::StarknetAuction;

// tapout new test
#[test]
fn test_audit_bid_event_issue() {
    let (auction_dispatcher, auction_contract, erc20_contract_address,
        _) = deploy_auction_contract();
    let erc20_dispatcher = IMockERC20TokenDispatcher { contract_address
        : erc20_contract_address };
    let bidder: ContractAddress = 123.try_into().unwrap();
    let mut spy = spy_events();

    // The owner calls start function.
    auction_dispatcher.start(86400, 10);

    // User (bidder) executes the bid process
    start_cheat_caller_address_global(bidder);

    erc20_dispatcher.mint(bidder, 15);
    erc20_dispatcher.token_approve(auction_contract, 15);

    // user calls the bid function with amount of 15.
    auction_dispatcher.bid(15);

    // check bidder 15 tokens has been transferred following the
       success of the bid process
    let bidder_balance = erc20_dispatcher.token_balance_of(bidder);
    assert(bidder_balance == 0, 'bidder balance should be zero');

    // wrong event bid amount which is still the previous setup bid
       amount at 10
    let wrong_event_bid_amount = StarknetAuction::Event::NewHighestBid(
       StarknetAuction::NewHighestBid {amount: 10, sender: bidder});

    // wrong event is indeed emitted, indicating the event isn't
       emitted with the latest bid amount
    spy.assert_emitted(@array![(auction_contract,
       wrong_event_bid_amount)]);

    stop_cheat_caller_address_global();
}
```

Step 3: Run the test `snforge test --features enable_for_tests test_audit_bid_event_issue`

```
...

Collected 1 test(s) from starknet_auction package
Running 0 test(s) from src/
Running 1 test(s) from tests/
[PASS] starknet_auction_integrationtest::test_contract::
    test_audit_bid_event_issue (gas: ~1962)
Tests: 1 passed, 0 failed, 0 skipped, 0 ignored, 16 filtered out
```

The test passes indicating that the event is wrongly emitted with previous setup bid instead of the user latest bid amount

Impact

Misleading information that confuse user and impact downstream/front-end process that relies on the event for rendering bid transaction information

Tools Used

Manual review with test

Recommendations

Make amendment to `StarknetAuction::bid` function by shifting the event emit after the update of `highest_bid`

```
fn bid(ref self: ContractState, amount: u64) {
    let time = get_block_timestamp();
    let erc20_dispatcher = IERC20Dispatcher { contract_address: self.
        erc20_token.read() };
    let sender = get_caller_address();
    let receiver = get_contract_address();
    let current_bid = self.highest_bid.read();

    assert(self.started.read(), 'Auction is not started');
    assert(time < self.bidding_end.read(), 'Auction ended');
    assert(amount > current_bid, 'The bid is not sufficient');

    self.bid_values.entry(sender).write(amount);
-   self.emit(NewHighestBid {amount: self.highest_bid.read(), sender:
    sender});
    self.highest_bidder.write(sender);
    self.highest_bid.write(amount);
+   self.emit(NewHighestBid {amount: self.highest_bid.read(), sender:
    sender});

    erc20_dispatcher.transfer(receiver, amount.into());
}
```