QuillAudits

Audit Report, September, 2024

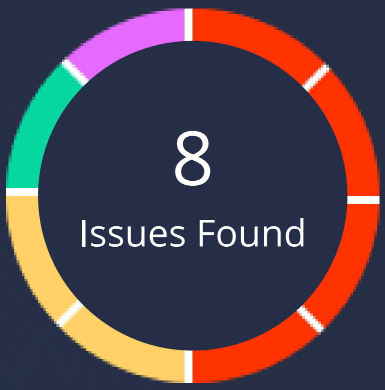
For

Table of Content

Executive Summary

Project Name	Bidds-Contracts-Coreum
Introduction	On Jul 14, 2024 -September 02 QuillAudits Team performed a security audit for Bidds-Contracts-Coreum smart contracts. The code for the audit was taken from following the official link: https://github.com/onxrp-insights/Bidds-Contracts-Coreum
Audit Scope	The scope of this audit was to analyze and document the Bidds-Contracts-Coreum codebase for quality, security, and correctness.
Review 1	14th July 2024
Commit Hash 1	ac272c2ebdcb435e34f240a073b4ca52cb89d467
Review 2	24th July 2024
Commit Hash 2	f47cd676b86d1ad1303792fa8663ca907c4327aa
Review 3	31st July 2024
Commit Hash 3	af030f5ac969042b4991ebb8eb9020fc7511b211
Review 4	2nd September 2024
Commit Hash 4	e78eba4d833f3e89752247412c8ba5580c65f603

Number of Issues per Severity



🟥 High	🟨 Medium
🟩 Low	🟪 Informational

	High	Medium	Low	Informational
Open Issues	0	0	0	0
Acknowledged Issues	1	1	0	0
Partially Resolved Issues	0	0	0	0
Resolved Issues	3	1	1	1

Checked Vulnerabilities

✓ Transaction-Ordering Dependence	✓ Outdated Dependencies
✓ Exception disorder	✓ Insufficient Randomness
✓ Balance equality	✓ Unsafe Arithmetic
✓ Unchecked math	✓ Unsafe Code Blocks
✓ Insecure Randomness	✓ Numerical precision errors
✓ Storage Exhaustion	✓ Business Logic Issues

Techniques and Methods

Throughout the audit of smart contracts, care was taken to ensure:

- The overall quality of code.
- Use of best practices.
- Code documentation and comments match logic and expected behavior.
- Liquidity computation on ranged ticks are correct

The following techniques, methods, and tools were used to review all the smart contracts.

Structural Analysis

In this step, we have analyzed the design patterns and structure of smart contracts. A thorough check was done to ensure the smart contract is structured in a way that will not result in future problems.

Static Analysis

A static Analysis of Smart Contracts was done to identify contract vulnerabilities. In this step, a series of automated tools are used to test the security of smart contracts.

Code Review / Manual Analysis

Manual Analysis or review of code was done to identify new vulnerabilities or verify the vulnerabilities found during the static analysis. Code base were completely manually analyzed, their logic was checked and compared with the one described in the whitepaper. Besides, the results of the automated analysis were manually verified.

Types of Severity

Every issue in this report has been assigned to a severity level. There are four levels of severity, and each of them has been explained below.

High Severity Issues

A high severity issue or vulnerability means that your smart contract can be exploited. Issues on this level are critical to the smart contract's performance or functionality, and we recommend these issues be fixed before moving to a live environment.

Medium Severity Issues

The issues marked as medium severity usually arise because of errors and deficiencies in the smart contract code. Issues on this level could potentially bring problems, and they should still be fixed.

Low Severity Issues

Low-level severity issues can cause minor impact and are just warnings that can remain unfixed for now. It would be better to fix these issues at some point in the future.

Informational

These are four severity issues that indicate an improvement request, a general question, a cosmetic or documentation error, or a request for information. There is low-to-no impact.

Types of Issues

Open

Security vulnerabilities identified that must be resolved and are currently unresolved.

Resolved

These are the issues identified in the initial audit and have been successfully fixed.

Acknowledged

Vulnerabilities which have been acknowledged but are yet to be resolved.

Partially Resolved

Considerable efforts have been invested to reduce the risk/impact of the security issue, but are not completely resolved.

High Severity Issues

1. Funds can be stolen using execute_accept_bid function

Line **Function - marketplace.execute.bids**

246
```
pub fn execute_accept_bid(
    deps: DepsMut<CoreumQueries>,
    env: Env,
    info: MessageInfo,
    id: OrderId,
    price: Coin,
) -> CoreumResult<ContractError> {
    nonpayable(&info)?;
    ....
}
```

Description

A bidder may potentially exploit the refund mechanism to illicitly obtain funds by manipulating the system while simultaneously engaging with counter-offers. This could occur if the bidder takes advantage of loopholes or vulnerabilities within the refund process to divert money for personal gain, all while continuing to participate in the negotiation process with counter-offers.

1. Bidder created a big for 10$ (Bidder transferred 10$)
2. Owner created a counter offer with 15$, in that counter the original Bid price was cloned as 10$
3. Bidder modifies its original bid and makes it to 1$, gets a refund of 9$.
4. Bidder then creates a counter offer of 2$, now I will have a refund of 8$ more and stole the money from the marketplace.

Remediation

To rectify this issue, the recommended steps involve fixing the refund mechanism.

Status

Resolved

2. Expired Listing

Description

When creating listing (list function) the expires_at is not being validated and it's possible to create expired listing.

Status

Resolved

3. Issue when Creating New Listing for token_1

Description

When we create a new listing for token_1 and there is an existing collection bid or bid for that collection but for token_2. It will accept either of those bids and will do the sale.

POC

This test is passing but should be failing

```
fn exploit_try_sell_into_bid() {
    let app = CoreumTestApp::new();
    let wasm: Wasm<'_, CoreumTestApp> = Wasm::new(&app);
    let bank = Bank::new(&app);
    let authz = Authz::new(&app);
    let start_time = Timestamp::from_seconds(app.get_block_time_seconds().unsigned_abs() +
MP_START_TIME);
    let accounts = setup_accounts::setup(&app, 5);
    let [owner, fee_recipient, minter, bidder, bidder_1] = &accounts[..] else {
        panic!("Expected 4 accounts");
    };
    // setup collection and nfts
    let col_data = setup_collection::setup_default_mp(&wasm, &accounts, 10);
    let mp_data = setup_marketplace::setup(&wasm, &accounts);
    let marketplace = mp_data.mp_address;
    let collection = &col_data.collection_addresses[0];
    let collection_id = format!("{}-{}", &col_data.msgs[0].symbol, collection).to_lowercase();
    let nfts = get_nfts_all(&wasm, &collection, &minter.address()).unwrap().nfts;
    let nft_id_1_1 = nfts[0].id.clone();
    let nft_id_1_2 = nfts[1].id.clone();
    app.increase_time(MP_START_TIME);
```

```
// Create bid
    let bid_res = add_bid(
        &wasm,
        &bidder,
        &marketplace,
        &collection_id,
        &nft_id_1_2,
        Some(start_time.plus_days(MIN_EXPIRY + 1)),
        false,
        &coin(150_000_000_000, NATIVE_DENOM),
        &coin(150_750_000_000, NATIVE_DENOM),
    );
    assert!(bid_res.is_ok());
    // Create listing and sell into bid
    grant_nft_msg_send(&authz, &minter, &collection_id, &nft_id_1_1,
&marketplace).unwrap();
    let owner_bal_before_accept = get_balance_amount(&bank, &minter.address());
    let royalty_bal_before_accept = get_balance_amount(&bank, &owner.address());
    let fee_bal_before_accept = get_balance_amount(&bank, &fee_recipient.address());
    let list_res = list_fixed(
        &wasm,
        &minter,
        &marketplace,
        &collection_id,
        &nft_id_1_1,
        Some(start_time.plus_seconds(MIN_EXPIRY + 1)),
        coin(100_000_000_000, NATIVE_DENOM),
        None,
    );
    assert!(list_res.is_ok());
    let sell_fee_paid = get_paid_fee(list_res.as_ref().unwrap());
    // Check money is transferred
    let owner_bal_after_accept = get_balance_amount(&bank, &minter.address());
    let royalty_bal_after_accept = get_balance_amount(&bank, &owner.address());
    let fee_bal_after_accept = get_balance_amount(&bank, &fee_recipient.address());
    assert_eq!(
        owner_bal_after_accept.sub(owner_bal_before_accept).add(sell_fee_paid),
        Uint128::from(91_500_000_000u128)
    );
    assert_eq!(
        royalty_bal_after_accept.sub(royalty_bal_before_accept),
        Uint128::from(8_000_000_000u128)
    );
    assert_eq!(fee_bal_after_accept.sub(fee_bal_before_accept),
Uint128::from(51_250_000_000u128));
    //contract balance should be zero
    let contract_balance = get_balance_amount(&bank, &marketplace);
    assert_eq!(contract_balance, Uint128::zero());
}
```

Status

Resolved

4. bug in execute_instant_sell

Description

When fetching the filtered collection bid for instant selling, it just takes the twice the length of token_ids that an owner is trying to instant sell. It doesn't check if those bids contains the specified token id or not.

Example

There is a filtered_col_bid that specifies a higher price for token_6. There exists 2 more filtered_col_bid which have a higher price for a different token lets say token_1. The filtered_col_bid with token_6 will be ignored. And in case another non-filtered col_bid that provides lower price for token_6 (or any token as its non-filtered) will be executed)

Status

Acknowledged

Bidds-Coreum Team's Comment

This is Intended Design,The problem here is that we cannot save the filtered collection bids by their token_id because that would be way too gas-expensive. (imagine making a collection offer to thousands of token_ids)

So what we are doing is keeping track of these filtered collection offers in our database, and sending matching collection offer ids along with the instant sell transaction (with suggested_orders). The check on the array with twice the length of the token_ids is a backup. However, I do think we need to take more filtered collection offers there as backup. (minimum of 30/40 or something)

Medium Severity Issues

5. Address is not being validated in function send

Line **Function - collection.execute.send**

246

```
pub fn send(deps: DepsMut<CoreumQueries>, info: MessageInfo, id: String,
receiver: String) -> CoreumResult<ContractError> {
    ...

    let msg = CoreumMsg::NFT(nft::Msg::Send {
        class_id: config.class_id.clone(),
        id: id.clone(),
        receiver,
    });
    ...
}
```

Description

The receiver address lacks validation to ensure it is a proper address, which contrasts with the validation performed in other functions. This omission introduces a potential vulnerability where invalid or malicious addresses could be processed, leading to unintended consequences.

Remediation

To rectify this issue, the recommended steps involve validating the receiver using resolve_address function as being done in other functions.

Status

Resolved

6.Potential Issue in Collection Bid

Description

There is a potential issue, if it's not intended. Based on the new change, we filter listing and only take the ones smaller than the minimum price sent in collection bid.

POC

```
@> let min_price = *col_bid
.prices
.as_ref()
.map(|p| p.iter().min())
.flatten()
.unwrap_or(&col_bid.price.amount);
let amount_paid = min_price + (col_bid.fee_used * min_price);


@> let max_list_price = amount_paid * (Decimal::percent(100) / (Decimal::percent(100) +
col_bid.fee_used));
let col_price = (col_bid.collection.clone(), col_bid.price.denom.clone());
@> let max = Some(Bound::exclusive((max_list_price.u128(), col_price.clone())));
let take_amount = col_bid.prices.as_ref().map_or(col_bid.nfts_amount_left, |_| 500u64) as
usize;


let mut listings = match &col_bid.token_ids {
Some(token_ids) => {
listings()
.idx
.collection_price
.sub_prefix(col_price)
@> .range(storage, None, max, StdOrder::Ascending)
.filter(|item| match item {
Ok((_, listing)) => token_ids.contains(&listing.token_id),
Err(_) => true,
})
.take(take_amount)
.map(|res| res.map(|item| item.1))
.collect::<StdResult<Vec<_>>>()?
}
None => listings()
.idx
.collection_price
.sub_prefix(col_price)
@> .range(storage, None, max, StdOrder::Ascending)
.take(take_amount)
.map(|res| res.map(|item| item.1))
.collect::<StdResult<Vec<_>>>()?,
};
```

Eg: we have 3 active listing at prices:
1)100
2)150
3)180

Now we make a collection Bid for Prices:
1)210
2)180
3)110

```rust
fn try_sell_into_listing() {
let app = CoreumTestApp::new();
let wasm: Wasm<'_, CoreumTestApp> = Wasm::new(&app);
let bank = Bank::new(&app);
let authz = Authz::new(&app);

let start_time = Timestamp::from_seconds(app.get_block_time_seconds().unsigned_abs() +
MP_START_TIME);

let accounts = setup_accounts::setup(&app, 4);
let [owner, fee_recipient, minter, bidder] = &accounts[..] else {
panic!("Expected 4 accounts");
};

// setup collection and nfts
let col_data = setup_collection::setup_default_mp(&wasm, &accounts, 10);
let mp_data = setup_marketplace::setup(&wasm, &accounts);
let marketplace = mp_data.mp_address;
let collection = &col_data.collection_addresses[0];
let collection_id = format!("{}-{}", &col_data.msgs[0].symbol, collection).to_lowercase();
let nfts = get_nfts_all(&wasm, &collection, &minter.address()).unwrap().nfts;
let nft_id_1_1 = nfts[0].id.clone();
let nft_id_1_2 = nfts[1].id.clone();
let nft_id_1_3 = nfts[2].id.clone();


app.increase_time(MP_START_TIME);


// Create listing
grant_nft_msg_send(&authz, &minter, &collection_id, &nft_id_1_1, &marketplace).unwrap();
grant_nft_msg_send(&authz, &minter, &collection_id, &nft_id_1_2, &marketplace).unwrap();
grant_nft_msg_send(&authz, &minter, &collection_id, &nft_id_1_3, &marketplace).unwrap();
let list_res = list_fixed(
&wasm,
&minter,
&marketplace,
&collection_id,
&nft_id_1_1,
Some(start_time.plus_seconds(MIN_EXPIRY)),
coin(100_000_000_000, NATIVE_DENOM),
None,);
let list_res_2 = list_fixed(
&wasm,
&minter,
&marketplace,
&collection_id,
&nft_id_1_2,
Some(start_time.plus_seconds(MIN_EXPIRY)),
coin(150_000_000_000, NATIVE_DENOM),
None,
);
let list_res_3 = list_fixed(
&wasm,
&minter,
&marketplace,
&collection_id,
&nft_id_1_3,
Some(start_time.plus_seconds(MIN_EXPIRY)),
coin(180_000_000_000, NATIVE_DENOM),
None,
);
assert!(list_res.is_ok());
assert!(list_res_2.is_ok());
assert!(list_res_3.is_ok());
// Create bid which is not sold into listing
let col_bid_res = add_col_bid(
&wasm,
&bidder,
&marketplace,
&collection_id,
Some(start_time.plus_seconds(MIN_EXPIRY)),
&coin(210_000_000_000, NATIVE_DENOM),
Some(vec![Uint128::from(210_000_000_000_u128),
Uint128::from(160_000_000_000_u128),
Uint128::from(110_000_000_000_u128)]),
&coin(487200000000, NATIVE_DENOM),
true,
);
assert!(col_bid_res.is_err());
match col_bid_res {
Err(e) => println!("Error :{:?}", e),
_ => println!("Worked")
}
}
```

Status

Acknowledged

Bidds-Coreum Team's Comment

While we should have used the max instead of the min price when the 'prices' are set, the test still wouldn't pass. It will take the lowest offer for the first price, lowest after that for the second etc. So we fixed the max price, but that still means that only the the first 2 prices are filled. But this is intended.

Low Severity Issues

5. Address is not being validated in function send

Line **Function - collection.execute.listing**

246

```
 pub fn list(
    deps: DepsMut<CoreumQueries>,
    env: Env,
    info: MessageInfo,
    sale_type: SaleType,
    collection: Collection,
    token_id: TokenId,
    params: ListInfo,
) -> CoreumResult<ContractError> {
    ...

let ListInfo {
        price,
        price_accept,
        funds_recipient,
        reserve_for,
        expires_at,
    } = params;


    ...
}
```

108

```
pub fn update_listing(
    deps: DepsMut<CoreumQueries>,
    env: Env,
    info: MessageInfo,
    id: OrderId,
    params: ListInfo,
) -> CoreumResult<ContractError> {
     ...
```

```
            }   let ListInfo {
                price,
                price_accept,
                funds_recipient,
                reserve_for,
                expires_at,
            } = params;
            ...
        }
```

Description

The expires_at value lacks validation to ensure its validity relative to the current time, which contrasts with the validation practices observed in other parts of the system. This absence introduces a potential risk where expired or improperly set timestamps could lead to incorrect decisions or actions within the application.

Another critical area to scrutinize carefully is when engaging in the creation and negotiation of counter- offers, as well as placing and updating bids within collections.

Remediation

To rectify this issue, the recommended steps involve introducing proper validations.

Status

Resolved

Informational Issues

8. Possibility of overflow and underflow when performing arithmetic operations

Line **Function - next_boost_id**

20

```
  pub fn next_boost_id(&mut self, storage: &mut dyn Storage) -> StdResult<u64>
{
      self.boost_id_count += 1;
      self.save(storage)?;
      Ok(self.boost_id_count)
  }
```

Line **Function - launchpad.contract.instantiate**

46

```
let max_id = msg.starting_id.unwrap() + u64::from(msg.token_amount.unwrap());
```

Description

In the current codebase, using native arithmetic operations raises the potential for overflow and underflow issues. To mitigate these risks, consider implementing checked safe methods for arithmetic operations. This oversight could lead to unpredictable behavior or errors if the value exceeds its maximum allowable value. It is crucial to implement safeguards to ensure that the counter does not overflow.

Remediation

Consider using the checked_add/checked_sub methods instead of the native arithmetic addition operation especially where typecasting is done from uint. This method provides built-in overflow checking, ensuring that if the addition operation exceeds the maximum value of the data type, an error or exception is raised

Status

Resolved

Automated Tests

Dylint

No major issues were found. Some false positive errors were reported by the tool. All the other issues have been categorized above according to their level of severity

Closing Summary

In this report, we have considered the security of the Bidds-Contracts-Coreum codebase. We performed our audit according to the procedure described above.

Some issues of High,Medium, Low and informational severity were found and Bidds-Coreum Team Resolved and Acknowledged the Issues

Disclaimer

QuillAudits Smart contract security audit provides services to help identify and mitigate potential security risks in Bidds-Contracts-Coreum smart contracts. However, it is important to understand that no security audit can guarantee complete protection against all possible security threats. QuillAudits audit reports are based on the information provided to us at the time of the audit, and we cannot guarantee the accuracy or completeness of this information. Additionally, the security landscape is constantly evolving, and new security threats may emerge after the audit has been completed.

Therefore, it is recommended that multiple audits and bug bounty programs be conducted to ensure the ongoing security of Bidds-Contracts-Coreum smart contracts. One audit is not enough to guarantee complete protection against all possible security threats. It is important to implement proper risk management strategies and stay vigilant in monitoring your smart contracts for potential security risks.

QuillAudits cannot be held liable for any security breaches or losses that may occur subsequent to and despite using our audit services. It is the responsibility of the Bidds-Contracts-Coreum Team to implement the recommendations provided in our audit reports and to take appropriate steps to mitigate potential security risks.

About QuillAudits

QuillAudits is a leading name in Web3 security, offering top-notch solutions to safeguard projects across DeFi, GameFi, NFT gaming, and all blockchain layers. With six years of expertise, we've secured over 1000 projects globally, averting over $30 billion in losses. Our specialists rigorously audit smart contracts and ensure DApp safety on major platforms like Ethereum, BSC, Arbitrum, Algorand, Tron, Polygon, Polkadot, Fantom, NEAR, Solana, and others, guaranteeing your project's security with cutting-edge practices.

1000+
Audits Completed

$30B
Secured

1M+
Lines of Code Audited

Follow Our Journey

Audit Report
September, 2024

For

QuillAudits

Canada, India, Singapore, UAE, UK

www.quillaudits.com

audits@quillhash.com