Blockchain Security Audit Report

Table Of Contents

1 Executive Summary

On 2023.05.10, the SlowMist security team received the team's security audit application for ord, developed the audit plan according to the agreement of both parties and the characteristics of the project, and finally issued the security audit report.

The SlowMist security team adopts the strategy of "white box" to conduct a complete security test on the project in the way closest to the real attack.

The test method information:

Test method	Description
Black box testing	Conduct security tests from an attacker's perspective externally.
Grey box testing	Conduct security testing on code modules through the scripting tool, observing the internal running status, mining weaknesses.
White box testing	Based on the open source code, non-open source code, to detect whether there are vulnerabilities in programs such as nodes, SDK, etc.

The vulnerability severity level information:

Level	Description
Critical	Critical severity vulnerabilities will have a significant impact on the security of the DeFi project, and it is strongly recommended to fix the critical vulnerabilities.
High	High severity vulnerabilities will affect the normal operation of the DeFi project. It is strongly recommended to fix high-risk vulnerabilities.
Medium	Medium severity vulnerability will affect the operation of the DeFi project. It is recommended to fix medium-risk vulnerabilities.
Low	Low severity vulnerabilities may affect the operation of the DeFi project in certain scenarios. It is suggested that the project party should evaluate and consider whether these vulnerabilities need to be fixed.
Weakness	There are safety risks theoretically, but it is extremely difficult to reproduce in engineering.
Suggestion	There are better practices for coding or architecture.

In black box testing and gray box testing, we use methods such as fuzz testing and script testing to test the robustness of the interface or the stability of the components by feeding random data or constructing data with a specific structure, and to mine some boundaries Abnormal performance of the system under conditions such as bugs or abnormal performance. In white box testing, we use methods such as code review, combined with the relevant experience accumulated by the security team on known blockchain security vulnerabilities, to analyze the object definition and logic implementation of the code to ensure that the code has the key components of the key logic. Realize no known vulnerabilities; at the same time, enter the vulnerability mining mode for new scenarios and new technologies, and find possible 0day errors.

2 Audit Methodology

The security audit process of SlowMist security team for smart contract includes two steps:

Smart contract codes are scanned/tested for commonly known and more specific vulnerabilities using automated analysis tools.

Manual audit of the codes for security issues. The contracts are manually analyzed to look for any potential problems.

Following is the list of commonly known vulnerabilities that was considered during the audit of the smart contract:

NO.	Audit Items	Result
1	Design Logic Audit	Some Risks
2	Others	Passed
3	State Consistency Audit	Some Risks
4	Failure Rollback Audit	Passed
5	Unit Test Audit	Passed
6	Integer Overflow Audit	Passed
7	Parameter Verification Audit	Passed

NO.	Audit Items	Result
8	Error Unhandle Audit	Passed
9	Boundary Check Audit	Passed
10	SAST	Some Risks

3 Project Overview

3.1 Project Introduction

Ord is an index, block explorer, and command-line wallet.

This audit focuses on checking whether the realization meets expectations against the following documents:

1. https://domo-2.gitbook.io/brc-20-experiment/
2. https://docs.ordinals.com/introduction.html

3.2 Coverage

Target Code and Revision:

https://github.com/okx/ord/tree/dev

Initial review commit: 97562216b9f61be396ae63b55257d95073a9f73c

Final review commit: 1257e4b92b11ce8d7c5f7e767ca668f5fcab1a96

https://github.com/okx/ord/tree/dev/src/brc20/updater.rs

https://github.com/okx/ord/tree/dev/src/index/updater/inscription_updater.rs

https://github.com/okx/ord/tree/dev/src/index/updater.rs

3.3 Vulnerability Information

The following is the status of the vulnerabilities found in this audit:

NO	Title	Category	Level	Status
N1	Supply chain security	SAST	Suggestion	Confirmed
N2	Block can have more than 20k inputs	Design Logic Audit	Suggestion	Confirmed
N3	Multi-threaded asynchronous processing of outputs may lead to sequential errors	State Consistency Audit	Low	Confirming
N4	Coinbase transactions may be used to construct inscriptions	Design Logic Audit	Low	Acknowledged
N5	Not detecting the relationship between self.height and block height	Design Logic Audit	Suggestion	Confirmed
N6	JSON extensibility leads to consensus fork risk	Design Logic Audit	Low	Acknowledged
N7	output_value cannot be equal to 0	Design Logic Audit	Low	Acknowledged
N8	Multiple legitimate inputs in a transaction may lead to consensus forking	Design Logic Audit	Low	Acknowledged
N9	Unspecified parameters may lead to consensus forking	Design Logic Audit	Low	Acknowledged
N10	Deploy `limit` may be out of range	Design Logic Audit	Low	Acknowledged

4 Findings

4.1 Visibility Description

The SlowMist Security team analyzed the visibility of major contracts during the audit, the result as follows:

4.2 Vulnerability Summary

[N1] [Suggestion] Supply chain security

Category: SAST

Content

```
Crate:     time
Version:   0.1.45
Title:     Potential segfault in the time crate
Date:      2020-11-18
ID:        RUSTSEC-2020-0071
URL:       https://rustsec.org/advisories/RUSTSEC-2020-0071
Severity:  6.2 (medium)
Solution:  Upgrade to >=0.2.23
Dependency tree:
time 0.1.45
└── chrono 0.4.24
    ├── rustls-acme 0.5.3
    │   └── ord 0.5.2
    ├── ord 0.5.2
    ├── diligent-date-parser 0.1.4
    │   └── atom_syndication 0.12.1
    │       └── rss 2.0.3
    │           └── ord 0.5.2
    └── atom_syndication 0.12.1
```

Solution

Crate `time` upgrade to >=0.2.23

Status

Confirmed

[N2] [Suggestion] Block can have more than 20k inputs

Category: Design Logic Audit

Content

- src/index/updater.rs

```
    // Not sure if any block has more than 20k inputs, but none so far after first
inscription block
```

```
    const CHANNEL_BUFFER_SIZE: usize = 20_000;
    let (outpoint_sender, mut outpoint_receiver) =
      tokio::sync::mpsc::channel::<OutPoint>(CHANNEL_BUFFER_SIZE);
    let (value_sender, value_receiver) = tokio::sync::mpsc::channel::<u64>
(CHANNEL_BUFFER_SIZE);
```

The maximum size of a transaction is close to the block size (4M), while the size of a transaction can be less than 200, which can theoretically exceed 20k inputs.

Solution

Increase CHANNEL_BUFFER_SIZE.

Status

Confirmed; CHANNEL_BUFFER_SIZE is the capacity of the channel, not affect business results.

[N3] [Low] Multi-threaded asynchronous processing of outputs may lead to sequential errors

Category: State Consistency Audit

Content

- src/index/updater.rs

```
std::thread::spawn(move || {
  let rt = tokio::runtime::Builder::new_multi_thread()
    .enable_all()
    .build()
    .unwrap();
  rt.block_on(async move {
    loop {
      let Some(outpoint) = outpoint_receiver.recv().await else {
        log::debug!("Outpoint channel closed");
        return;
      };
      // There's no try_iter on tokio::sync::mpsc::Receiver like
std::sync::mpsc::Receiver.
      // So we just loop until BATCH_SIZE doing try_recv until it returns None.
      let mut outpoints = vec![outpoint];
      for _ in 0..BATCH_SIZE-1 {
        let Ok(outpoint) = outpoint_receiver.try_recv() else {
          break;
        };
        outpoints.push(outpoint);
      }
      // Break outpoints into chunks for parallel requests
```

```
        let chunk_size = (outpoints.len() / PARALLEL_REQUESTS) + 1;
        let mut futs = Vec::with_capacity(PARALLEL_REQUESTS);
        for chunk in outpoints.chunks(chunk_size) {
          let txids = chunk.iter().map(|outpoint| outpoint.txid).collect();
          let fut = fetcher.get_transactions(txids);
          futs.push(fut);
        }
        let txs = match try_join_all(futs).await {
          Ok(txs) => txs,
          Err(e) => {
            log::error!("Couldn't receive txs {e}");
            return;
          }
        };
        // Send all tx output values back in order
        for (i, tx) in txs.iter().flatten().enumerate() {
          let Ok(_) =
value_sender.send(tx.output[usize::try_from(outpoints[i].vout).unwrap()].value).await
else {
            log::error!("Value channel closed unexpectedly");
            return;
          };
        }
      }
    })
});
```

The `new_multi_thread()` function creates a runtime that can run multiple OS threads, which means that asynchronous tasks in this runtime can run concurrently, taking advantage of the power of multi-core processors.

However, the completion time of each thread in the concurrent runtime is not always the same, and the transaction with the next highest order is called `value_sender.send` first, causing an error in the order.

Solution

Use a type like `map` instead of `vector` to record the results of asynchronous runs.

Status

Confirming

[N4] [Low] Coinbase transactions may be used to construct inscriptions

Category: Design Logic Audit

Content

- src/index/updater.rs

```
if let Some((tx, txid)) = block.txdata.get(0) {
self.index_transaction_sats(
  tx,
  *txid,
  &mut sat_to_satpoint,
  &mut coinbase_inputs,
  &mut sat_ranges_written,
  &mut outputs_in_block,
  &mut inscription_updater,
  index_inscriptions,
)?;
}
```

the coinbase transaction in Bitcoin does indeed have specific format rules. It's the first transaction in every new block and it creates new bitcoins and pays them to the miner as a reward. Here are some format restrictions on a coinbase transaction:

- Input count: A coinbase transaction has only one input.
- Previous output index: The previous output index (previous output index) that the input of a coinbase transaction references has to be set to 0xFFFFFFFF.
- Previous transaction ID: The previous transaction ID (previous transaction id) that the input of a coinbase transaction references has to be set to 32 bytes of 0.
- Unlocking script (Signature Script): The unlocking script (also known as the signature script or scriptSig) of the input of a coinbase transaction must contain specific data. The first four or five bytes have to be a special value which is the height of the block. After this, miners can place arbitrary data into the unlocking script, but the total length of this data has to be between 2 bytes to 100 bytes.
- Outputs: The outputs of a coinbase transaction contain the newly created bitcoins, which are paid to the miner. The quantity of the newly created bitcoins is determined by Bitcoin's monetary policy, which states that the quantity of new bitcoins created halves every 210,000 blocks. Additionally, the outputs of a coinbase transaction can also contain bitcoins that the miner collected from transaction fees.

Some additional transaction inputs and outputs may be considered legitimate if these rules are met, so coinbase transactions cannot be ignored.

Solution

Defensive programming is recommended and can also be effective in detecting when coinbase is used to construct inscriptions

Status

Acknowledged; Inscriptions are considered only for index 0 in the input.

[N5] [Suggestion] Not detecting the relationship between self.height and block height

Category: Design Logic Audit

Content

- src/index/updater.rs

```
for (txid, brc20_transaction) in inscription_collects {
  brc20_action_count +=
    brc20_updater.index_transaction(self.height, block.header.time, txid,
brc20_transaction)?
      as u64;
}

statistic_to_count.insert(&Statistic::LostSats.key(), &lost_sats)?;

statistic_to_count.insert(&Statistic::BRC20ActionCount.key(), &brc20_action_count)?;

height_to_block_hash.insert(&self.height, &block.header.block_hash().store())?;

self.height += 1;
self.outputs_traversed += outputs_in_block;
```

Solution

Here you can add a defensive programming, self.height==block.bip34_block_height()

Status

Confirmed

[N6] [Low] JSON extensibility leads to consensus fork risk

Category: Design Logic Audit

Content

- src/index/updater/inscription_updater.rs

```
#L130:
deserialize_brc20_operation(Inscription::from_transaction(&inscribe_tx).unwrap())
#L184:
if let Ok(operation) = deserialize_brc20_operation(inscription.unwrap()) {
```

The json in the inscription will be parsed here.

```
pub fn deserialize_brc20(s: &str) -> Result<Operation, JSONError> {
  let value: Value = serde_json::from_str(s).map_err(|_| JSONError::InvalidJson)?;
  if value.get("p") != Some(&json!(PROTOCOL_LITERAL)) {
    return Err(JSONError::NotBRC20Json);
  }

  Ok(serde_json::from_value(value).map_err(|e|
JSONError::ParseOperationJsonError(e.to_string()))?)
}
```

The serde_json::from_value() function in Rust will indeed ignore fields in the JSON data that are not defined in the target type.

When deserializing with the Serde library in Rust, if a field is not defined in your target type but exists in the input JSON data, this field is ignored.

If some extra fields are added to a legitimate brc20 json format, it can be parsed normally, but this behavior may be a consensus violation and may create a consensus fork.

PoC like this:

```
{
"p":"brc-20",
"op":"deploy",
"tick":"web9",
"max":"21000000",
"lim":"1000",
"ex-field-1":"1",
"ex-field-2":"100"
}
```

Solution

Need for clearer community consensus

Status

Acknowledged

[N7] [Low] output_value cannot be equal to 0

Category: Design Logic Audit

Content

- src/index/updater/inscription_updater.rs

```
let mut output_value = 0;
for (vout, tx_out) in tx.output.iter().enumerate() {
  let end = output_value + tx_out.value;

  while let Some(flotsam) = inscriptions.peek() {
    if flotsam.offset >= end {
      break;
    }

    let new_satpoint = SatPoint {
      outpoint: OutPoint {
        txid,
        vout: vout.try_into().unwrap(),
      },
      offset: flotsam.offset - output_value,
    };

    let flotsam = inscriptions.next().unwrap();
    self.update_inscription_location(input_sat_ranges, flotsam, new_satpoint)?;

    if let Some(inscription_data) = inscriptions_collector
      .iter_mut()
      .find(|key: &&mut (u64, InscriptionData)| {
        key.1.inscription_id == flotsam.clone().inscription_id
      })
      .map(|value| &mut value.1)
    {
      let action = &mut inscription_data.action;
      action.set_to(Some(tx_out.script_pubkey.clone()));
      inscription_data.action = action.clone();
      inscription_data.new_satpoint = Some(new_satpoint);
    }
```

```
  }

  output_value = end;

  self.value_cache.insert(
    OutPoint {
      vout: vout.try_into().unwrap(),
      txid,
    },
    tx_out.value,
  );
}
```

Solution

Check if `output_value` is greater than 0

Status

Acknowledged; If output is 0, then the inscription will be transferred to the miner, which is legal on consensus.

[N8] [Low] Multiple legitimate inputs in a transaction may lead to consensus forking

Category: Design Logic Audit

Content

- src/index/updater/inscription_updater.rs

```
let mut input_value = 0;
for tx_in in &tx.input {
  if tx_in.previous_output.is_null() {
    input_value += Height(self.height).subsidy();
  } else {
    for (old_satpoint, inscription_id) in
      Index::inscriptions_on_output(self.satpoint_to_id, tx_in.previous_output)?
    {
      inscriptions.push(Flotsam {
        offset: input_value + old_satpoint.offset,
        inscription_id,
        origin: Origin::Old(old_satpoint),
      });

      let inscribe_satpoint = SatPoint {
        outpoint: OutPoint::new(inscription_id.txid, inscription_id.index),
        offset: 0,
      };
```

```
      if !is_coinbase {
        if old_satpoint == inscribe_satpoint {
          let inscribe_tx = if let Some(t) =
self.tx_cache.remove(&inscription_id.txid) {
            t
          } else {
            self
              .index
              .get_transaction_with_retries(inscription_id.txid)?
              .ok_or(anyhow!(
                "failed to get inscription transaction for {}",
                inscription_id.txid
              ))?
          };
          if let Ok(_) =

deserialize_brc20_operation(Inscription::from_transaction(&inscribe_tx).unwrap())
          {
            inscriptions_collector.push((
              input_value + old_satpoint.offset,
              InscriptionData {
                txid,
                inscription_id,
                old_satpoint,
                new_satpoint: None,
                action: Action::Transfer(TransferAction {
                  from_script: inscribe_tx
                    .output
                    .get(0)
                    .ok_or(anyhow!("faild to index output for {}",
inscription_id.txid))?
                    .script_pubkey
                    .clone(),
                  to_script: None,
                }),
              },
            ))
          }
        };
      }
    }

    input_value += if let Some(value) =
self.value_cache.remove(&tx_in.previous_output) {
      value
    } else if let Some(value) = self
      .outpoint_to_value
      .remove(&tx_in.previous_output.store())?
    {
```

```
      value.value()
    } else {
      self.value_receiver.blocking_recv().ok_or_else(|| {
        anyhow!(
          "failed to get transaction for {}",
          tx_in.previous_output.txid
        )
      })?
    }
  }
}
```

There may be more than one legal input in the transaction.

Solution

When a brc20 is successfully obtained, it should break or return to stop continuing to detect the input of the current transaction, the subsequent inputs may not be consensus.

Status

Acknowledged; Inscriptions are considered only for index 0 in the input.

[N9] [Low] Unspecified parameters may lead to consensus forking

Category: Design Logic Audit

Content

- src/brc20/updater.rs

```
let dec = Num::from_str(&deploy.decimals.map_or(MAX_DECIMAL_WIDTH.to_string(), |v|
v))?
  .checked_to_u8()?;
if dec > MAX_DECIMAL_WIDTH {
  return Err(Error::BRC20Error(BRC20Error::InvalidDecimals(dec)));
}
let base = BIGDECIMAL_TEN.checked_powu(dec as u64)?;

let supply = Num::from_str(&deploy.max_supply)?;

if supply > Into::<Num>::into(u64::MAX) {
  return Err(Error::BRC20Error(BRC20Error::InvalidMaxSupply(supply)));
}
```

There is no mention of a maximum `decimals` of 18 in the reference #1, which may lead to consensus forking.

There is no mention of a maximum `max` of `u64::MAX` in the reference #1, which may lead to consensus forking.

Solution

Need for clearer community consensus

Status

Acknowledged

[N10] [Low] Deploy `limit` may be out of range

Category: Design Logic Audit

Content

- src/brc20/updater.rs

```
if limit.sign() == Sign::NoSign
  || limit > Into::<Num>::into(u64::MAX)
  || limit.scale() > dec as i64
{
  return Err(Error::BRC20Error(BRC20Error::MintLimitOutOfRange(
    lower_tick.as_str().to_string(),
    limit,
  )));
}
```

Theoretical `limit` must <= `supply`.

Solution

```
if limit.sign() == Sign::NoSign
  || limit > Into::<Num>::into(u64::MAX)
  || limit.scale() > dec as i64
  || limit > supply
{
  return Err(Error::BRC20Error(BRC20Error::MintLimitOutOfRange(
    lower_tick.as_str().to_string(),
    limit,
  )));
}
```

Status

Acknowledged; In other implementations, `restrict > supply` is allowed, such as unisat.

5 Audit Result

Audit Number	Audit Team	Audit Date	Audit Result
0X002305120001	SlowMist Security Team	2023.05.10 - 2023.05.12	Passed

Summary conclusion: The SlowMist security team use a manual and SlowMist team's analysis tool to audit the project, during the audit work we found 7 low risk, 3 suggestion vulnerabilities.

6 Statement

SlowMist issues this report with reference to the facts that have occurred or existed before the issuance of this report, and only assumes corresponding responsibility based on these.

For the facts that occurred or existed after the issuance, SlowMist is not able to judge the security status of this project, and is not responsible for them. The security audit analysis and other contents of this report are based on the documents and materials provided to SlowMist by the information provider till the date of the insurance report (referred to as "provided information"). SlowMist assumes: The information provided is not missing, tampered with, deleted or concealed. If the information provided is missing, tampered with, deleted, concealed, or inconsistent with the actual situation, the SlowMist shall not be liable for any loss or adverse effect resulting therefrom. SlowMist only conducts the agreed security audit on the security situation of the project and issues this report. SlowMist is not responsible for the background and other conditions of the project.

Official Website

www.slowmist.com

E-mail

team@slowmist.com

Twitter

@SlowMist_Team

Github

https://github.com/slowmist