InterBTC - Vault Client

Technical audit report prepared for Interlay

Reference	**22-09-1042-REP**
Version	**1.0**
Date	**2022/11/21**

Quarkslab SAS
10 boulevard Haussmann
75009 Paris
France

Contents

1 Project Information

Document history			
Version	**Date**	**Details**	**Authors**
1.0	2022/11/21	Initial Version	Robin David and another Quarkslab auditor[1].

Quarkslab		
Contact	**Role**	**Contact Address**
Frédéric Raynal	CEO	fraynal@quarkslab.com
Stavia Salomon	Sales Manager	ssalomon@quarkslab.com
Matthieu Ramtine Tofighi Shirazi	Project Manager	mrtofighishirazi@quarkslab.com
Robin David	R&D Engineer	rdavid@quarkslab.com
Quarkslab auditor[1]	R&D Engineer	

Interlay		
Contact	**Role**	**Contact Address**
Dominik Harz	Co-Founder & CTO	dominik@interlay.io
Alexei Zamyatin	Co-Founder & CEO	alexei@interlay.io
Sander Bosma	Software Engineer	sander@interlay.io
Gregory Hill	Research & Software Engineering	gregory@interlay.io

[1] Anonymized for public release of the report.

2 Executive Summary

This report describes the results of the security evaluation made by Quarkslab on the `vault-client` component developed by Interlay. It's the off-chain component of the bridging solution developed by the company. The vault is a client of both InterBTC/Kintsugi and Bitcoin chains. It performs the issuing and redeeming actions, and ensures that assets are properly locked in one chain before releasing them on the other.

The audit aims at verifying that the implementation respects the specification, that private keys managed are held securely and that no security issues can impact availability or leading to resources theft. Audits have already been performed on this component by another audit company [1]. Issues have also been found in the past via the Immunefi bug bounty program [2, 3].

This security assessment highlighted four potential weaknesses. One of them is ranked high as it can potentially cause a double payment when a vault restarts. However, conditions to meet in order to trigger these issues are very unlikely to happen if not negligible in terms of probabilities. Indeed, it would require specific events to happen on Bitcoin chain or the inability of the vault to reach the `bitcoin-core` node at very specific moments. Finding summary is available in Table 2.2 and details in Chapter 5.

This document in version 1.0, was delivered to Interlay on 2022/11/21, for public release.

2.1 Disclaimer

This report reflects the work and results obtained within the duration of the audit on the specified scope (see. Section 3.3) as agreed between Interlay and Quarkslab. Tests are not guaranteed to be exhaustive and the report does not ensure the code to be bug or vulnerability free.

2.2 Findings Summary

The severity classification, informative, low, and medium, reflects a relative hierarchy between the various findings of this report (cf. Table A in appendix). It depends of the threat model considered and security properties considered. In this context, availability is mandatory but not critical while double-spending or incorrect minting of assets is. Table A below summarizes issues and defects found during the audit.

ID	Description	Recommendation	Impact
`HIGH_01`	Potential double redeem payment due to ignored errors.	Do not proceed if the Bitcoin node request returns an error.	`High`
`LOW_01`	Undetected issue payment.	Changing the `find_map` function call by an iteration of the whole map.	`Low`
`LOW_02`	Potential panic in the Bitcoin block relayer.	Changing the `unwrap` not to panic on failure if the block does not exist.	`Low`

INFO_01	Unsafe arithmetic in various uses of sum in the light client.	Changing the `sum` by explicit saturating and overflowing operations.	Info

3 Context and Scope

3.1 Context

Interlay aims at bridging Bitcoin assets to Polkadot, Kusama networks and at larger extend substrate-based blockchains.

By default blockchains are isolated networks where each node of a given blockchain is collaborating following a given set of rules. The purpose of a bridge is to lock an asset on the originating blockchain and to mint (or release) an asset, equivalent in value, on the destination blockchain. The main difficulty is the great discrepancies between the blockchain technologies to bridge. Indeed, the underlying language (virtual machine), the block time, the cryptographic primitives can differ. As an example, while substrate-based blockchains are extremely customizable and flexible, the Bitcoin blockchain has very limited programmability capabilities.

To develop its solution, Interlay has a parachain on the Kusama network named Kintsugi. The organization entered the Polkadot auction in the second batch and won the 10^{th} parachain slot. Their interBTC parachain on Polkadot started on the 11^{th} of March 2022.

Interlay bridge is completely permissionless unlike some bridges operated by private companies or operated by a reduced set of operators using a multi-signature mechanism. As such, anyone can participate in the bridge effort by bringing collateral and running vaults. These vaults are the off-chain components acting as clients of both chains to bridge. They register themselves via an extrinsic on the parachain and can manage theirs on-chain actions via this interface.

The whole design has been defined in the `xclaim` [4] publication and is thoroughly discussed in Interlay documentation [5].

3.2 Safety and Security Properties

The component reviewed is not an on-chain component, so it is not directly exposed to block production issues. However, it needs to cope with all peculiarities of the two technologies it is bridging. That also implies synchronization and peg issues that are critical for a bridge to behave correctly. Indeed, the main security concern is ensuring the 1:1 peg value hold thanks to the collateral provided. Some of the issues and attack scenarios considered are the following:

- double payments / replay attacks *(on either of the two chains)*;
- blockchain forks;
- griefing *(locking collateral assets by issuing invalid bogus requests)*.

The security model considered takes into account a misbehaving user of any of the two chains, attempting to attack available interfaces and the idiosyncratic behavior of blockchains.

Price manipulation and oracle issues are attack vectors left out of scope for this audit.

3.3 Scope

The scope of this audit is narrowed down to the code of the vault client rust modules. The audited source code is available on the repository `interbtc-clients` [1] on Github.

By means of dependencies the vault client uses the following internal modules that are part of the audit:

- `vault/` main service, running the whole vault;
- `bitcoin/` implementing the BitcoinCore RPC client;
- `runtime/` parachain client;
- `service` utility functions.

The usage of libraries and frameworks not explicitly written by Interlay are left out of the scope of this audit. As consequence, libraries left out of scope are:

- `bitcoincore-rpc`;
- `tokio`;
- `subxt`;
- `esplora-btc-api`;
- all substrate dependencies;
- *(and multiple other minor dependencies)*.

More information about the specific version and the audit setup can be found in the Section 3.4.

3.4 Audit Settings

As the Interlay codebase is under active development, the version used for the audit has been frozen in agreement with the Interlay team. Exact versions and commit ID are shown in Table 3.1. Also, the network used on the Bitcoin side is **regtest**. On the parachain side, the network used is Kintsugi in **testnet**.

Project	interbtc-clients
Repository	`https://github.com/interlay/interbtc-clients`
Version	1.16.2
Commit hash	`e5c2a95ff124d923c019ed915f7554f268a60505`
Commit date	2022/09/12

Table 3.1: interbtc-clients version references

[1] `https://github.com/interlay/interbtc-clients/`

Kusama parachain is configured with parameters mentioned in table 3.2. The main difference with the Rococo configuration used in this assessment is the confirmation time set to 1.

Symbol	KINT
Decimals	12
SS58 prefix	2092
Bitcoin confirmation	6
Parachain confirmation	300

Table 3.2: Configuration of Kintsugi parachain

3.4.1 Addendum October 13, 2022

During the audit, some newly developed features have been added in the scope of the audit. These features are:

- auto-updater client wrapper for downloading and running new vault versions [code];
- pidfile mechanism to prevent running duplicate vault with the same configuration [code];
- light client, that introduce `electrs` [6] as interface with the Bitcoin node [code].

For these features, the machine running is considered genuine and third-party components used like `reqwest` [7] are assumed to properly check TLS certificates for HTTP communications.

3.5 Runtime configuration

The audit has been performed on a combination of `kusama-rococo` and `bitcoin-regtest` to ease testing. The parachain was launched using `polkadot-launch`[2], and BitcoinCore v22.0.0 for the Bitcoin side. Atop, `electrs` 0.4.1 has been used. The `polkadot-launch` configuration and launching command for the other components are documented in Annex B.

Finally, as part of the audit setting, a dedicated communication channel has been created on Discord enabling Quarkslab auditors and Interlay engineers to interact in a fast and effective manner.

3.6 Methodology

The methodology employed to audit the bridge can be summarized in the following steps.

Codebase Familiarization, Node Setup. The first step of the audit is digging into the ecosystem, technologies taking place (Bitcoin, substrate, XClaim) and reading the documentation at

[2]The project `zombienet` [8] [9] is meant to replace `polkadot-launch` but it was not working because of [284].

our disposal. Then one shall create a working setup of the product in testnet. In this case it requires setting up a valid parachain (producing blocks), a valid Bitcoin node and a vault doing the interaction in between the two. Quarkslab used various resources for that end [5, 10].

Static Code Review and Analysis. The next step consisted in statically reviewing the code both manually and using some code analysis tooling (e.g `clippy` [11]). That includes looking for standard weak code constructs in Rust like unsafe arithmetic, unsafe unpacking or panics. As part of that step, a global reckoning of components and dependencies is also being performed. The feasibility of some attack scenarios or logical bugs is studied according to the code available.

Dynamic Testing. If required some dynamic tests have been performed, to validate the feasibility of some scenarios either by writing custom rust tests or directly on the system as it is running *(on rococo and regtest)*. It's also the occasion to test in practice some assertion about the code behavior identified statically.

4 Vault Static Code Review

This chapter aims at given a brief overview of the inner organization and working of the code vault and more especially its interactions with both chains. Issues and defect are described in the next Chapter 5.

4.1 Vault Service & Asynchronous Tasks

At startup, the vault first derives keys used for the parachain and Bitcoin. Then, it starts the prometheus exporter and creates a PID file to detect duplicate vault instances on the same chain id. Finally, it connects sequentially to both chains and launch the vault service. This service implements a trait called `Service` and which implementation is located in *system.rs*.

The service heavily relies on `tokio` [12] to run asynchronous tasks. At startup the vault launches the following tasks which many of them are monitoring parachain events defined in the parachain runtime. The tasks are the following:

- *fee estimation*: catch the `FeedValues` event generated by the parachain.
- *redeem request*: catch the `RequestRedeem` event of the parachain and execute it on the Bitcoin blockchain.
- *issue request*: catch the `RequestIssue` event, checks that it is addressed to the vault and if so, emit an internal `Opened` event to get the request processed by another task.
- *issue execute*: catch the parachain `ExecuteIssue` event. That retrieves when an issue request as successfully been executed.
- *issue cancel*: catch the `CancelIssue` event to remove the issue from the pending ones.
- *issue cancel scheduler*: listen issuing events and cancel them if they expired without being executed in order to unlock collateral.
- *replace request*: catch `RequestReplace` event, for a vault to be replaced by another one.
- *accept request*: catch the `AcceptReplace` event, if a vault accepted to replace another one.
- *execute replace*: catch the `ExecuteReplace` event directed to the vault and transfers the associated bitcoins.
- *replace cancel scheduler*: monitors the current replace queries.
- *refund request*: catch the `RequestRefund` event and process them *(removed in more recent vault version)*.
- *parachain block*: catch the `UpdateActiveBlock` to keep track of newly created parachain blocks.
- *bitcoin block*: catch the `StoreMainChainHeader` event of the parachain generated when a block header is submitted by the btc-relay.

- *vaultid registration*: catch the `RegisterVault` event triggered when new vaults are being registered.
- *bitcoin relay*: active task listening for finalized Bitcoin blocks. It submits them to the parachain.
- *issue executor*: monitor transactions on Bitcoin chain to catch issuing ones thanks to their identifier.
- *metrics*: promotheus related tasks.
- *restart timer*: periodic task that restarts the vault.

The behavior of these tasks has been reviewed as part of the audit.

4.2 Parachain Interactions

Interactions with the parachain node are being performed using `subxt` [13] developed by Parity Technologies. As mentioned in the previous section the vault monitor and react to the events generated on the parachain. That is the idiomatic way extrinsics submitted on the parachain will trigger events on the various vaults. In terms of extrinsic submission, they are also performed through `subxt`. Here are some of the extrinsic that can be submitted by vaults:

- `issue.executeIssue()`
- `issue.cancelIssue()`
- `replace.acceptReplace()`
- `replace.executeReplace()`
- `redeem.executeRedeem()`
- `oracle.feedValues()`
- `btcRelay.initialize()`
- `btcRelay.storeBlockHeader()`
- `vaultRegistry.registerVault()`
- `vaultRegistry.depositCollateral()`
- `vaultRegistry.withdrawCollateral()`
- `vaultRegistry.registerPublicKey()`

4.3 Bitcoin Interactions

In the vault, interaction with a `bitcoin-core` node is performed on the RPC endpoint with `rust-bitcoincore-rpc` [14]. Alternatively it can interact with an `electrs` endpoint instead.

The latter is expected to become the recommended way to set up a vault. Interlay uses the Blockstream fork[1].

Note

The blockstream version of `electrs` is a fork of another version[a] which is way ahead in terms of commit. Vault setup documentation does note recommend any of the two versions.

[a] https://github.com/romanz/electrs

The Bitcoin blockchain is monitored by the *issue executor* and *btc relay* tasks mentioned in the first section. The issue executor retrieved all finalized blocks and iterate transactions to find issuing one according to its issue set.

On the transaction submission side, they all goes through the `create_and_send_transaction` of the `BitcoinCoreApi` trait. The function create the transaction with the vault key and send it the Bitcoin node through RPC.

```
async fn create_and_send_transaction(
  &self,
  address: Address,
  sat: u64,
  fee_rate: SatPerVbyte,
  request_id: Option<H256>,
) -> Result<Txid, Error> {
  let tx = self.create_transaction(address, sat, fee_rate, request_id).await?;
  let txid = self.send_transaction(tx).await?;
  Ok(txid)
}
```

Bitcoin transactions are submitted in the redeem process by `listen_for_redeem_requests` function and in the replacement process by `listen_for_accept_replace`. These interactions have also been reviewed as part of the audit process.

[1] https://github.com/Blockstream/electrs

5 Recommendations

5.1 Vault functionality

The vault functionality is the primary job of the vault client. It keeps track of the pending issues, redeem and replace requests and tries to fullfil them. This is a very sensitive piece of software as it handles tokens on both chains.

5.1.1 Multiple Issue payment in a single Bitcoin transaction

LOW_01	Undetected issue payment (Bitcoin transaction)		
Category	Logic bug		
Status	Present		
Rating	Severity: Low	Impact: Low	Exploitability: None

Description: If a user makes multiple issue requests and pays them in the same Bitcoin transaction, then the second payment output will be missed by the vaults. This is due to the use of `find_map` in the `process_transaction_and_execute_issue` function which only returns the first matching element as stated in its documentation.

```
let addresses: Vec<BtcAddress> = transaction
  .extract_output_addresses()
  .into_iter()
  .filter_map(|payload| BtcAddress::from_payload(payload).ok())
  .collect();
let mut issue_requests = issue_set.lock().await;
if let Some((issue_id, address)) = addresses.iter().find_map(|address| {
  let issue_id = issue_requests.get_key_for_value(address)?;
  Some((*issue_id, *address))
}) {...}
```

> Applies function to the elements of iterator and returns the first non-none result. `iter.find_map(f)` is equivalent to `iter.filter_map(f).next()`.

Recommandation: A possible fix for this issue would be turning the `if let` construct into a `while let` construct. However this alone should not be enough as the core of the function can return and needs to be changed accordingly.

Exploitability: This issue was assessed as not exploitable, as neither Quarkslab's auditors and Interlay's team found a way to abuse this bug, and the user is able to submit the transaction proof himself so this issue was rated as low severity.

Note

This issue is tracked on Github as issue#424 [a]

[a] https://github.com/interlay/interbtc-clients/issues/424

5.1.2 Double payment at vault startup and restart

HIGH_01	Double payment due to ignored errors		
Category	Error handling		
Status	Fixed		
Rating	Severity: High	Impact: Critical	Exploitability: None

Description: During a vault startup or restart, it retrieves all the pending requests, including redeem and replace requests already paid but not yet reported to the parachain due to a lack of confirmations. The vault client then creates a stream to retrieve the transactions on the Bitcoin chain, both in the `mempool` and the latest blocks. The transactions in the stream are then processed to retrieve payment for the pending requests. If a transaction fails to be retrieved because of an RPC error from the Bitcoin client, that transaction is ignored and the next one is processed, which is the core of the issue because any request without payment will then be paid once all transactions have been processed or ignored, including paid requests whose payment was missed and will be paid again.

```
  // iterate through transactions in reverse order, starting from those in the mempool, and
  // gracefully fail on encountering a pruned blockchain
  let mut transaction_stream = bitcoin::reverse_stream_transactions(&read_only_btc_rpc, btc_start_heigh
↪  t).await?;
  while let Some(result) = transaction_stream.next().await {
    let tx = match result {
      Ok(x) => x,
      Err(e) => {
        tracing::warn!("Failed to process transaction: {}", e);
        continue;
      }
    };

    // get the request this transaction corresponds to, if any
    if let Some(request) = get_request_for_btc_tx(&tx, &open_requests) {
      // remove request from the hashmap
      open_requests.retain(|&key, _| key != request.hash);

      <snippet>
      // start a new task to (potentially) await confirmation and to execute on the parachain
      // make copies of the variables we move into the task
      let parachain_rpc = parachain_rpc.clone();
      let btc_rpc = vault_id_manager.clone();
      spawn_cancelable(shutdown_tx.subscribe(), async move {
        let btc_rpc = match btc_rpc.get_bitcoin_rpc(&request.vault_id).await {
          Some(x) => x,
          None => {
            tracing::error!(
```

```
                "Failed to fetch bitcoin rpc for vault {}",
                request.vault_id.pretty_print()
              );
              return; // nothing we can do - bail
            }
          };
            <snippet>
      });
    }
  }

  // All requests remaining in the hashmap did not have a bitcoin payment yet, so pay
  // and execute all of these
  for (_, request) in open_requests {
    // there are potentially a large number of open requests - pay and execute each
    // in a separate task to ensure that awaiting confirmations does not significantly
    // delay other requests
    spawn_cancelable(shutdown_tx.subscribe(), async move {
      <snippet>
      match request
        .pay_and_execute(parachain_rpc, vault, num_confirmations, auto_rbf)
        .await
      {
        <snippet>
      }
    });
  }
```

This behavior was reproduced in the following code snippet to highlight the issue.

```
use futures::{prelude::*, stream::StreamExt};
use rand::prelude::*;

#[derive(Debug)]
pub enum Error {
  NotInMemPool,
  RPCError,
}

#[tokio::main]
async fn main() {
  if let Err(e) = vault_run_service().await {
    println!("Aborting because of {:?}", e);
    println!("The vault actually triggers a restart");
    std::process::exit(1);
  }
  std::process::exit(0);
}

const FAIL_CHANCE:u32 = 20;

async fn vault_run_service() -> Result<(), Error> {
  let mut things_we_want_to_process: Vec<u32> = (0..40).collect();
  let mut transaction_stream = make_a_stream().await?;
  while let Some(result) = transaction_stream.next().await {
    let tx = match result {
      Ok(x) => x,
      Err(e) => {
        println!("Failed to process transaction: {:?}", e);
        continue;
      }
    };
    println!("Next, no error so we see the transaction {}", tx);
    things_we_want_to_process.retain(|x| *x!= tx);
```

```
    // here we will retrieve the payment in the TX and wait
    // for its inclusion, otherwise we will try to pay after this loop
  }
  println!("We missed payment if it was in one of those transactions :
↪  {:#?}", things_we_want_to_process);
  println!("Now we do pay the pending redeem and replace, even the already paid ones if they were in
↪  a missed transaction");
  Ok(())
}

pub async fn make_a_stream() -> Result<impl Stream<Item = Result<u32, Error>> + Unpin, Error> {
  let mempool_blocks = stream::iter(make_an_iterator(FAIL_CHANCE).await?);
  let included_blocks = stream::iter(make_an_iterator(FAIL_CHANCE + 5).await?);
  let mut rng = rand::thread_rng();
  let rnd_num: u32 = rng.gen_range(0..101);
  // RPC errors can also happen here and trigger a restart
  if rnd_num < FAIL_CHANCE {
    return Err(Error::RPCError);
  }
  Ok(mempool_blocks.chain(included_blocks))
}

pub async fn make_an_iterator(
  chance: u32,
) -> Result<Box<dyn Iterator<Item = Result<u32, Error>> + Send>, Error> {
  println!("Making the iterator with chance {}%", chance);
  println!("A chance of {}% is for fake mempool transactions", FAIL_CHANCE);
  let nums: Vec<u32> = if chance == FAIL_CHANCE {
    (0..20).collect()
  } else {
    (20..40).collect()
  };

  let iterator = nums.into_iter().filter_map(move |num| {
    match can_fail_randomly(num, chance) {
      Ok(x) => Some(Ok(x)),
      Err(Error::NotInMemPool) => None, // not in mempool anymore, so filter out
      Err(Error::RPCError) => Some(Err(Error::RPCError)),
    }
  });
  Ok(Box::new(iterator))
}

fn can_fail_randomly(num: u32, chance: u32) -> Result<u32, Error> {
  let mut rng = rand::thread_rng();
  let rnd_num: u32 = rng.gen_range(0..101);
  if rnd_num == 100 {
    return Err(Error::NotInMemPool);
  }
  if rnd_num < chance {
    return Err(Error::RPCError);
  }
  return Ok(num);
}
```

Recommandation: The easiest way to fix this issue would be to consider errors in this step as deadly instead of ignoring them. This is somewhat dangerous to ignore these errors. Furthermore, the stream usage makes it hard to run the failed query again, if possible at all.

Another way to fix it can be to track issues which have been previously paid by the vault on disk (e.g in a file or database). However this could still be sensitive to race conditions and would require extreme care.

Note

This issue was fixed by Interlay in pull request #414 [a]. Only considering network errors as deadly should fix the issue without introducing the risk of making it possible to run a DoS against the client.

[a] https://github.com/interlay/interbtc-clients/pull/414

Warning

The definition of network errors should be updated prior to the official release of the light client in order to fully fix this issue. This is tracked in issue#369 [a]

[a] https://github.com/interlay/interbtc-clients/issues/369

Exploitability: This issue was assessed as not exploitable as it would require being able to trigger a restart of the victim vault and cut the communication with the local Bitcoin node in a timely manner. An attacker with such capabilities shall already be able to steal funds in a much simpler and reliable manner.

Note

Despite this issue not being exploitable, Quarkslab's auditors rated it as high as it weakens the financial safety of the bridge, even though the events triggering this bug have a low chance to happen.

5.2 Relay functionality

The relay functionality is the secondary job of the vault client. It keeps track of the Bitcoin blocks to relay them to the bridge. This functionality is optional. Yet, at least one vault among the registered ones have to do it for the whole bridge to work.

5.2.1 Panic in the relayer

LOW_02	Potential panic in the relayer		
Category	Panic		
Status	Present		
Rating	Severity: Low	Impact: Medium	Exploitability: None

Description: When the relayer tries to send multiple Bitcoin blocks to the parachain, it makes use of the `collect_headers` function which could potentially panic due to the use of `unwrap()`.

When the relayer queries the Bitcoin node using the `get_block_header` function, the `collect_headers` function could receive a `Ok(None)` result which would trigger a panic.

```
async fn collect_headers(height: u32, batch: u32, cli: &impl Backing) -> Result<Vec<Vec<u8>>, Error> {
  let mut headers = Vec::new();
  for h in height..height + batch {
    headers.push(cli.get_block_header(h).await.map(|header| header.unwrap())?);
  }
  Ok(headers)
}

async fn get_block_header(&self, height: u32) -> Result<Option<Vec<u8>>, Error> {
  let block_hash = match BitcoinCoreApi::get_block_hash(self, height).await {
    Ok(h) => h,
    Err(BitcoinError::InvalidBitcoinHeight) => {
      return Ok(None);
    }
    Err(err) => return Err(err.into()),
  };
  let block_header = BitcoinCoreApi::get_block_header(self, &block_hash).await?;
  Ok(Some(serialize(&block_header)))
}
```

This behavior can be observed using Interlay's test suite. By changing the return value of the `DummyBacking::get_block_header` function to `Ok(None)`, the test `submit_next_with_1_confirmation_batch_submission_succeeds` fails because it panics.

Recommandation: This issue can be fixed by changing the line pushing the header using a if let construct, such as:

```
if let Some(header) = cli.get_block_header(h).await? { headers.push(header)};
```

Exploitability: However, analysis of the code and thorough discussions with the Interlay team showed that this issue cannot be triggered on purpose. It was assessed that this panic should only trigger in case a fork happens around the time the Bitcoin difficulty is adjusted. It would then require that a branch with more work but fewer blocks wins.

Given the inability to trigger this bug, acting as a relayer is optional, and the very unlikely setup required for it to trigger, Quarkslab's team rated this issue as low.

Note

This issue is tracked on Github as issue#423 [a]

[a] `https://github.com/interlay/interbtc-clients/issues/423`

Warning

The same issue could happen in the `submit_next` function, during the initialization process of the issuing chain.

```
if !self.issuing.is_initialized().await? {
  let start_height = self.start_height.unwrap_or(self.get_num_confirmed_blocks().await?);
  tracing::info!("Initializing at height {}", start_height);
  self.issuing
    .initialize(
      self.backing.get_block_header(start_height).await?.unwrap(),
      start_height,
    )
    .await?;
}
```

5.3 Bitcoin light client

5.3.1 Unsafe arithmetic

INFO_01	Various use of sum in the light client		
Category	Unsafe arithmetic		
Status	Present		
Rating	Severity: Info	Impact: N/A	Exploitability: N/A

Description: The sum function in Rust is a utility function to compute the sum of the elements of a Vec. This function uses the default Add function which can overflow.

This function is used in multiple places in the light client, such as the fee_rate function.

```
async fn fee_rate(&self, txid: Txid) -> Result<SatPerVbyte, BitcoinError> {
  let tx = self.get_transaction(&txid, None).await?;
  let vsize = tx.weight().div_ceil(WITNESS_SCALE_FACTOR) as u64;
  let recipients_sum = tx.output.iter().map(|tx_out| tx_out.value).sum::<u64>();

  let inputs = try_join_all(tx.input.iter().map(|input| async move {
    <snippet>
  }))
  .await?;
  let input_sum = inputs.iter().sum::<u64>();
  <snippet>
}
```

Recommandation: The use of the sum function should be prohibited in favor of explicit saturating or overflowing operations, despite the added boilerplate code.

Exploitability: This issue is not likely to be exploitable and relates more to best practices, being fully explicit on the semantic of operations performed and coherence with the other client implementation which does not use this function.

Warning

There are also multiple use of unsafe arithmetic functions (addition, multiplication, division) all over the Bitcoin light client, but no specific case was identified as problematic. Nonetheless, it might be dangerous so the same recommendation applies to be consistent with the rest of the codebase.

6 Conclusion

The code size of the vault is reasonable but heavily relies on asynchronous tasks that must properly interact with two different blockchains. These blockchains do use different technologies and have different consensus and finality rules. Thus, assessing the vault required careful code review. The audit was performed for a duration of slightly more than 55 days.

First of all, the quality of the interBTC code base managed by Interlay should be highlighted once again. Some issues found can theoretically be very problematic because leading to double payment. However, the conditions to meet in order to trigger the bug are very unlikely, if not infeasible in practice. As such, no significant issues have been found during the audit. Thanks to the dedicated discord channel created for the security assessment, issues have been disclosed to the team which also have been of great help to better assess the effective impact of these issues. Given the criticality of the project, Quarkslab encourages Interlay to keep the code quality consistency for the whole project and to continue having their code audited by external companies.

Glossary

extrinsic State changes that come from the outside world, i.e. they are not part of the system itself. Extrinsics can take two forms, "inherents" and "transactions".

interBTC is Interlay's flagship product - Bitcoin on any blockchain. A 1:1 Bitcoin-backed asset, fully collateralized, interoperable, and censorship-resistant. interBTC will be hosted as a Polkadot parachain and connected to Cosmos, Ethereum and other major DeFi networks.

Bibliography

[1] Josef Widder, Cezara Dragoi, and Shon Feder. *InterBTC Parachain Modules and Vault Client: Protocol Design and Source Code*. [report]. Sept. 9, 2021. URL: https://github.com/informalsystems/audits (cit. on p. 2).

[2] Interlay. *Kintsugi Released Urgent Security Patches*. URL: https://medium.com/interlay/kintsugi-released-urgent-security-patches-aebf969ee087 (visited on Nov. 10, 2022) (cit. on p. 2).

[3] PwningEth. *The Defrauded Fraud Proof of A Bitcoin Bridge*. URL: https://pwning.mirror.xyz/jlT8OgtwN3mQf3KdYmXdcSXbE4s95JzT3eR3wxiLmpw (visited on Nov. 10, 2022) (cit. on p. 2).

[4] Alexei Zamyatin et al. "XCLAIM: Trustless, Interoperable, Cryptocurrency-Backed Assets". In: *2019 IEEE Symposium on Security and Privacy (SP)*. 2019, pp. 193–210. DOI: 10.1109/SP.2019.00085 (cit. on p. 4).

[5] *Interlay & Kintsugi Documentation*. Interlay. URL: https://docs.interlay.io/ (visited on Mar. 14, 2022) (cit. on pp. 4, 7).

[6] paritytech. *zombienet*. URL: https://github.com/paritytech/zombienet (visited on Nov. 10, 2022) (cit. on p. 6).

[7] Sean McArthur. *reqwest*. URL: https://github.com/seanmonstar/reqwest (visited on Nov. 10, 2022) (cit. on p. 6).

[8] paritytech. *zombienet*. URL: https://github.com/paritytech/zombienet (visited on Nov. 10, 2022) (cit. on p. 6).

[9] paritytech. *polkadot-launch*. URL: https://github.com/paritytech/polkadot-launch (visited on Nov. 10, 2022) (cit. on p. 6).

[10] *interBTC Technical Specification*. v5.6.1. Interlay. URL: https://spec.interlay.io/ (visited on Mar. 14, 2022) (cit. on p. 7).

[11] rust-lang. *rust-clippy*. URL: https://github.com/rust-lang/rust-clippy (visited on Mar. 14, 2022) (cit. on p. 7).

[12] tokio. *tokio*. URL: https://tokio.rs (visited on Nov. 10, 2022) (cit. on p. 8).

[13] paritytech. *subxt*. URL: https://github.com/paritytech/subxt (visited on Nov. 10, 2022) (cit. on p. 9).

[14] rust-bitcoin. *rust-bitcoincore-rpc*. URL: https://github.com/rust-bitcoin/rust-bitcoincore-rpc (visited on Nov. 10, 2022) (cit. on p. 9).

Appendix A

Severity Classification

Severity	Description
Critical	Exploitable major issues that could result in loss of funds or DDoS attack.
High	Exploitable major issues that could result in loss of funds or DDoS attack and whose setup has high requirements. Also issues that are likely to become exploitable and whose exploitation would be trivial.
Medium	Medium issues that cannot be directly exploited, such as the use of unsafe arithmetic or potential panics. These issues could potentially lead to loss of funds or DDoS attacks in future updates.
Low	Low issues that cannot be directly exploited such as mismatch between the specification and implementation on pre/post conditions or incorrect weight. These issues could potentially lead to logic bugs and cheap computation or storage.
Info	Diverse informative recommendations on code structure, documentation, TODO annotations, etc.

Appendix B

The configuration file used to launch the parachain is the following:

```
{
    "relaychain": {
        "bin": "/home/vagrant/binaries/polkadot-v0.9.15",
        "chain": "rococo-local",
        "nodes": [
            {"name": "alice", "wsPort": 9944, "port": 30444},
            {"name": "bob", "wsPort": 9955, "port": 30555},
            {"name": "charlie", "wsPort": 9966, "port": 30666}
        ],
        "genesis": {
            "runtime": {
                "runtime_genesis_config": {
                    "configuration": {
                        "config": {
                            "validation_upgrade_frequency": 1,
                            "validation_upgrade_delay": 20}}}}
        }
    },
    "parachains": [
        { "bin": "./interbtc-parachain",
          "id": "2121",
          "balance": "1000000000000000000000",
          "nodes": [
                { "wsPort": 9988,
                  "port": 31200,
                  "name": "alice",
                  "flags": ["--",    "--execution=wasm"]}
          ],
          "chain": "rococo-local-2000"
        }
    ],
    "simpleParachains": [],
    "hrmpChannels": [],
    "types": {},
    "finalization": false
}
```

Figure B.1: `polkadot-launch configuration file`

On the bitcoin side, the daemon is launched with the following command.

```
./bitcoin-22.0/bin/bitcoind -regtest \
    -server \
    -rpcuser=$USER \
    -rpcpassword=$PASS \
    -walletrbf=1 \
    -zmqpubrawtx=tcp://127.0.0.1:29000 \
    -zmqpubhashtx=tcp://127.0.0.1:29000 \
    -zmqpubrawblock=tcp://127.0.0.1:29000 \
    -zmqpubhashblock=tcp://127.0.0.1:29000
```

We launch the node with publishers of blocks and transactions for dynamic tests monitoring the bitcoin chain. Then `electrs` just need to be launched to get a valid setup.

```
    ./target/release/electrs -vvvv --daemon-dir $HOME/.bitcoin --network regtest
↪   --cookie $USER:$PASS
```

Last a vault private key is generated with *subkey generate --output-type json > keyfile.json*. Then the vault can be run by providing the IP address of both the parachain node and the bitcoin-core node. Note that for the registration to work, enough `KINT` token have to be sent to the address so that it can properly submit extrinsics allowing to register itself as a vault with some collateral.

```
./target/release/vault --bitcoin-rpc-url http://localhost:18443
                       --bitcoin-rpc-user $USER
                       --bitcoin-rpc-pass $PASS
                       --keyfile keyfile.json
                       --keyname 0x92aff85590a666773[snip]7d39955297be4049
                       --auto-register=KSM=1000000
                       --btc-parachain-url ws://localhost:9988
```