Supplier Server
Magic Protocol
Security Assessment

Oinspect

Magic



Magic Supplier Server Security Assessment

V220906

Prepared for Stacker Labs • August 2022

- 1. Executive Summary
- 2. Assessment and Scope
- 3. Summary of Findings
- 4. Detailed Findings
- SUP-1 Supplier funds can be stolen
- SUP-2 Unchecked result of finalization transaction might lead to xBTC loss
- SUP-3 Inconsistent state through race conditions
- SUP-4 Funds at risk upon transaction broadcasting contingency
- SUP-5 Information disclosure on HTTP error response
- SUP-6 Supplier server dependency on Render cloud
- SUP-7 Environment variables poorly validated
- SUP-8 Funds at risk due to Bitcoin constant fees
- 5. Disclaimer

1. Executive Summary

In **July 2022**, **Magic** engaged Coinspect to perform a source code review of the Supplier Server. The goal of the project was to evaluate the security of the application, which is a critical off-chain component of its bridge. The Supplier Server allows liquidity suppliers (BTC and xBTC) to automate swap operations.

High Risk	Medium Risk Low Risk	
1	3	0
Fixed 0	Fixed 3	Fixed 0

Coinspect identified a single high-risk issue caused by the possibility of a swapper stealing funds from a supplier when the size of BTC transactions exceeds 1024 bytes on outbound swaps.

Medium-risk issues are related to the possibility of a supplier losing xBTC if an outbound swap finalization fails. Similarly, a supplier could lose BTC if the server fails to broadcast transactions. Lastly, multiple race conditions that could lead to state inconsistencies have been detected throughout the code.

2. Assessment and Scope

The audit started on July 11 and was conducted on git repository at https://github.com/magicstx/supplier-server as of commit 48d419f8c8934b087640bca79389464bdcf72add of July 8 from the branch main.

The audited files have the following sha256 hash:

```
d7717c3f5f8f7f3b1cea934f669359fbac634c855011aeb7075b9e1319b666af
                                                                  ./routes/bull-adapter.ts
34a91bc9478baf03a167ba9809f7907d30025b2a9a931e0cd51c5950a33f914f
                                                                  ./stacks.ts
1a0e2db6a102b6d90f914322bc1b3c70d1a0ab6acd52ede62db06d25be7846ad
                                                                  ./stacks-api.ts
65765c313c1ea30d7edbbe53f267ace9384461b5ebace25e6c478412537a70bc
                                                                  ./events.ts
5a584d024c53c1b9f5871ee6144bb2ee1dbef4b33d191a7480504ade4f676728
                                                                  ./clarigen/deployments/simnet.ts
582523bceb5dae92bb359de491d3038cdb14816ab005ac8f256d8570fa453ac2
                                                                  ./clarigen/deployments/testnet.ts
43a4e578b403abc5e5dbdd70f25fb3dbd12ec5b13b3865c080dcbdddb01c6a37
                                                                  ./clarigen/deployments/devnet.ts
2022e887a2dc8c7e9e90fe1a40af6ca1bd6f25c406a04795b8ad5dc82fbe4ebc
                                                                  ./clarigen/index.ts
83d1ccd7f21cd46ecea8d5823fbfac33f498a0ec9398c1dd14aa4e022afbe600
                                                                  ./utils.ts
e26f8672ecbf02aea129ad5092ea0abef8ea5f68ba87b487a042b141607b5d67
                                                                  ./index.ts
f28ebbd95b4be1fe2f2fbae8e092eea6491a63cc588bae9d60736a1994e209f6
                                                                  ./processors/outbound.ts
506b180d2ad27cdfc8d65155eed066b7286c71d20ec64db7807f3527f1ca699e
                                                                  ./processors/finalize-outbound.ts
0fd1b65b06be11aa8301e6f72174c0624e1e8f872097fe624a614755e23c67a8
                                                                  ./processors/redeem-htlc.ts
10efaa2acd52cdb0c228abef779982fdc25437532050451e83cb0d79a6bb07a0
                                                                  ./types/electrum-client.d.ts
f70512781bbd0ea63535246fca3cd5d7234cfd1c54f1b1426d9fcb6276ab6959
                                                                  ./logger.ts
6ee7c5ed160046ae2ac473f907c36887c690066e783fd193926dc463078535f1
                                                                  ./config.ts
a6097fd0c2f1a191b03fb4c762d5c68693be08a0c5c0914f36ad944d84453476
                                                                  ./wallet.ts
147eb0d6cd36773144a372df1dcdb617c5e8dbd677e7eb3c462c828a9959c25a
                                                                  ./worker/queues.ts
a9734a9d4af364cffd9b2851be4942c611b374b5ca2c32db1e88f4579d4b1156
                                                                  ./worker/index.ts
61e4f722c8fbd217bead1639015367f539cab693a3a023ef48ca90773b0e7762 ./store.ts
```

The Magic Bridge allows users (swappers) to exchange BTC->xBTC (inbound swaps) and xBTC->BTC (outbound swaps) on the Magic Protocol. Swappers can choose which supplier they want to operate with. On the other hand, suppliers are BTC and xBTC liquidity providers that pool xBTC in the Magic Bridge contract on the Stacks chain. Finally, the Supplier Server automates suppliers' swap operations.

As for inbound swaps, swappers send the corresponding BTC directly to suppliers using Hash Time Locked Contracts (HTLC) transactions. Once the swapper submits proof of the HTLC transaction to the Magic Bridge contract, it escrows the supplier xBTC funds already pooled in the contract. Then, after swappers provide the HTLC preimage, the Magic Bridge contract releases the xBTC and emits an event containing the preimage so that the supplier can redeem the locked BTC.

On the other hand, once a swapper initiates an outbound swap, the Magic Bridge contract locks up its xBTC and emits an event which is handled by the supplier. The

supplier then sends BTC directly to the swapper and later sends proof of the BTC transaction to the Magic Bridge contract. If the proof is correct, the contract transfers xBTC owned by the swapper back to the supplier xBTC pool.

This assessment focused mainly on the correctness of inbound and outbound event processors, comprising the topics such as the integrity of the swap process, behaviors upon crash or reset, denial-of-service caused by malformed transactions or events, the handling of transaction results, and the impossibility to reprocess past events, to name a few.

The overall Supplier Server code was easy to follow, although no tests were provided in the audited code. The documentation, made available in the Magic Protocol site was complete and straightforward. On the other hand, Coinspect noticed a few TODO comments throughout the code, one of which represents a risk to suppliers (see SUP-8). Make sure the code has no pending work before releasing it to production.

Regarding the lack of test cases, as the project depends on the correct interaction and information retrieval from the Magic Protocol contract and the Bitcoin network, not having a unit test suite nor integration tests increases the risk of undetected bugs and vulnerabilities. It is therefore heavily recommended to implement a complete and thorough test suite.

Coinspect noticed that the Supplier Server project is intended to be executed on the Render cloud only, although the server can be hosted on other platforms. The documentation should alert users about the risk of running the Supplier Server outside of the Render cloud, such as using a Redis database without disk persistence which can lead to the loss of funds. Consider also supporting different cloud platforms and formats by adding the required configs. This will remove the dependency between Supplier Servers and the Render cloud services.

Another threat related to the operation of the Supplier Server, which is not a vulnerability itself, is the possibility of suppliers losing pooled xBTC when Supplier Servers are offline. In the event of an offline Supplier Server for an extended period, if there were any inbound swap events left unprocessed (unclaimed), swappers would be able to claim back the transferred BTC from the HTLC transaction causing

suppliers to lose funds. Suppliers should be made aware of this threat and how to prevent it.

Lastly, a few undocumented preconditions are required to run a Supplier Server, such as the setup of Redis or the format of environment variables, as reported below. Additionally, the server relies on the connection to Bitcoin and Stacks nodes in sync. However, there is no attempt to detect and react to a situation where nodes are out of sync, which could lead to the loss of funds. Validating the connection and status of the Stacks and Bitcoin nodes will increase the resilience of the code.

3. Summary of Findings

ld	Title	Total Risk	Fixed
SUP-1	Supplier funds can be stolen	High	√
SUP-2	Unchecked result of finalization transaction might lead to xBTC loss	Medium	~
SUP-3	Inconsistent state through race conditions	Medium	•
SUP-4	Funds at risk upon transaction broadcasting contingency	Medium	~
SUP-5	Information disclosure on HTTP error response	Info	~
SUP-6	Supplier server dependency on Render cloud	Info	√
SUP-7	Environment variables poorly validated	Info	~
SUP-8	Funds at risk due to Bitcoin constant fees	Info	V

4. Detailed Findings



Description

A malicious swapper can steal funds from any supplier through carefully manipulating the supplier's wallet.

The bridge contract supports Bitcoin transactions of up to 1024 bytes, as shown in the following function signature:

The supplier wallet, on the contrary, does not have any consideration on the size of the outbound swap transaction. The functions selectCoins and sendBtc do not present any check for the transaction size.

Therefore, the swapper can force the supplier to use too many UTXOs when performing an outbound transaction. Once that happens, the swapper receives the BTC in the Bitcoin blockchain. Then, when the supplier tries to prove the transaction using the finalize-outbound function it will fail due to the size of the transaction being more than 1024 bytes. If the supplier fails to finalize the swap within OUTBOUND_EXPIRATION (200 blocks), the swapper can revoke the swap and thus receive the escrowed xBTC back (plus the BTC previously transferred by the supplier server).

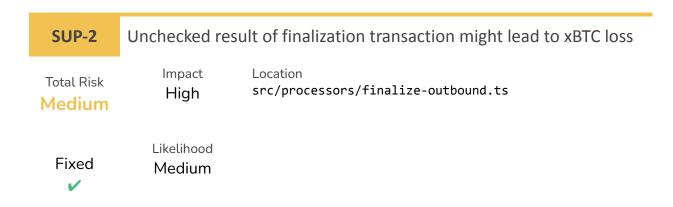
If the supplier does not hold enough small UTXOs to spend, the swapper can provide those UTXO first by multiple small inbound swap operations.

Recommendation

Make sure that transactions signed by the supplier are accepted by the bridge.

Status

Partially fixed. The supplier server now exits if the transaction is more than 1024 bytes long in the reviewed version with commit c721f728401f266a738d88d1a233d79798b77038 from the feat/develop branch. However, this issue is not yet entirely fixed since the supplier server does not automatically consolidate UTXOs.



Supplier servers fail to successfully finalize outbound swaps under certain conditions, without checking the result of the finalization transaction. This would allow swappers to claim back escrowed xBTC and still receive BTC sent by suppliers.

The supplier server code fails to check the result of the finalizeOutboundSwap transaction and considers the outbound swap as finalized. Below is shown the code snippet corresponding to the finalizeOutbound function.

```
const stxTxid = await withElectrumClient(async client => {
 const data = await txData(client, txid);
 const finalizeTx = bridge.finalizeOutboundSwap(
  data.block,
  data.prevBlocks,
  data.tx,
  data.proof,
  Øn,
  id
 );
 const receipt = await provider.tx(finalizeTx, { nonce });
 return receipt.txId; //Magic txiD
}):
log.debug({ stxTxid }, `Submitted finalize outbound Stacks tx: ${stxTxid}`);
await removePendingFinalizedOutbound(client, id, txid);
await setFinalizedOutbound(client, id, stxTxid);
return true;
```

However, the transaction could fail due to multiple reasons. For instance, in the event of a Bitcoin fork not yet registered in Stacks, the Merkle proof submitted by the supplier server would be considered invalid.

Then, once OUTBOUND_EXPIRATION has elapsed (200 blocks), the swapper can revoke the swap and thus receive the escrowed xBTC back (plus the BTC previously transferred by the supplier server).

Recommendation

Check the result of the swap finalization transaction. If it's not successful, attempt finalizing the swap a few blocks later.

Status

Fixed in the reviewed version with commit c721f728401f266a738d88d1a233d79798b77038 from the feat/develop branch. The logic has been refactored to have a more robust failure monitoring regarding outbound finalization

SUP-3 Inconsistent state through race conditions Total Risk Medium Impact Location src/processors/finalize-outbound.ts src/worker/index.ts

Fixed Likelihood Medium

~

Description

Multiple race conditions throughout the source code may lead to inconsistent states. The inconsistent states have some unpredictable outcomes, such as processing swap events and sending funds twice.

Example 1:

The txData function called when finalizing outbound swaps computes the block height where the transaction was included:

```
const tx = await client.blockchain_transaction_get(txid, true);
const burnHeight = await confirmationsToHeight(tx.confirmations);
```

The blockchain height is later retrieved in the confirmationsToHeight function and used along the tx information to compute the burnHeight:

```
export async function confirmationsToHeight(confirmations: number) {
  const nodeInfo = await fetchCoreInfo();
  const curHeight = nodeInfo.burn_block_height;
  const height = curHeight - confirmations + 1;
  return height;
}
```

However, it can happen that nodeInfo.burn_block_height increases right after calling client.blockchain_transaction_get. Therefore, the burnHeight used would be incorrect.

Example 2:

The getContractEventsUntil function returns new events that occurred since the last seen transaction ID, sorted from oldest to newest. This function might also return duplicate events if new Stacks blocks appear during the recursion call of the getContractEventsUntil function.

```
void eventCronQueue.process(1, async () => {
  const lastSeenTxid = await getLastSeenTxid(client);
  const newEvents = await getContractEventsUntil(lastSeenTxid);
  if (newEvents.length > 0) {
    const topics = newEvents.map(e => e.print.topic);
    logger.debug({ topic: 'processEvents', topics }, `Processing ${newEvents.length} new events`);
  }
  const [firstEvent] = newEvents;
  if (firstEvent! == undefined) {
    await setLastSeenTxid(client, firstEvent.txid);
  }
  const eventJobs = newEvents.map(event => ({ data: { event: serializeEvent(event) } }));
  await eventQueue.addBulk(eventJobs);
  return {
    newEvents: newEvents.length,
    lastSeenTxid,
  };
});
```

When executing the asynchronous function processOutboundSwap, duplicate events would be processed with little time difference. In case the time difference is too low, both events would pass the shouldSendOutbound function and potentially send a duplicate outbound payment.

Recommendation

Prevent any possible inconsistency of this type.

Status

Example 1: in case there's a mismatch between height/header an error will be triggered and the transaction will be retried following the fix added in SUP-2. Example2: Fixed in the reviewed version with commit c721f728401f266a738d88d1a233d79798b77038 from the feat/develop branch. The server now records the 'chain tip' at the start of the pagination, as well as after. If the tips don't match, the job silently exits.

Total Risk Medium Funds at risk upon transaction broadcasting contingency Location Src/wallet.ts:135 Likelihood

Description

Fixed

Low

The supplier server code does not retry sending transactions upon errors when broadcasting them. Therefore, suppliers could miss claiming BTC from an HTLC transaction due to an unsent transaction.

The redeem function in src/processors/redeem-htlc.ts calls the tryBroadcast function, which sends a transaction to release BTC from an inbound swap HTLC transaction as follows:

```
export async function redeem(txid: string, preimage: Uint8Array) {
 return withElectrumClient(async client => {
  psbt.finalizeInput(0, (index, input, script) => {
     const partialSigs = input.partialSig;
     if (!partialSigs) throw new Error('Error when finalizing HTLC input');
     const inputScript = bScript.compile([
       partialSigs[0].signature,
       Buffer.from(preimage),
      opcodes.OP TRUE,
     ]);
     const payment = payments.p2sh({
       redeem: {
         output: script,
         input: inputScript,
     });
     return {
      finalScriptSig: payment.input,
      finalScriptWitness: undefined,
  });
   const final = psbt.extractTransaction();
  const finalId = final.getId();
   await tryBroadcast(client, final);
   const btcAmount = satsToBtc(swap.sats);
  logger.info(
     { redeemTxid: finalId, txUrl: getBtcTxUrl(finalId), htlcTxid: txid, amount: swap.sats },
     Redeemed inbound HTLC for ${btcAmount} BTC`
  return finalId;
 });
```

}

The result of this call is not properly handled in the redeem function nor in upper calls, and the event is lost from the finalizeInboundQueue. Thus, the supplier server does not retry redeeming BTC from an HTLC transaction. Below is the code corresponding to the tryBroadcast function.

```
export async function tryBroadcast(client: ElectrumClient, tx: Transaction) {
 const id = tx.getId();
 try {
   await client.blockchain_transaction_broadcast(tx.toHex());
   const amount = tx.outs[0].value;
   logger.info(
       topic: 'btcBroadcast',
       txid: id,
       txUrl: getBtcTxUrl(id),
       amount,
      Broadcasted BTC tx ${id}`
   );
   return id;
 } catch (error) {
   logger.error({ broadcastError: error, txId: id }, `Error broadcasting: ${id}`);
if (typeof error === 'string' && !error.includes('Transaction already in block chain')) {
     if (error.includes('Transaction already in block chain')) {
       logger.debug(`Already broadcasted redeem in ${id}`);
       return;
     if (error.includes('inputs-missingorspent')) {
       logger.debug(`Already broadcasted redeem in ${id}`);
     }
   }
   await client.close();
   throw error;
}
```

Therefore, an unexpected network error that impedes the redeem transaction broadcasting might allow a swapper to claim back sent funds.

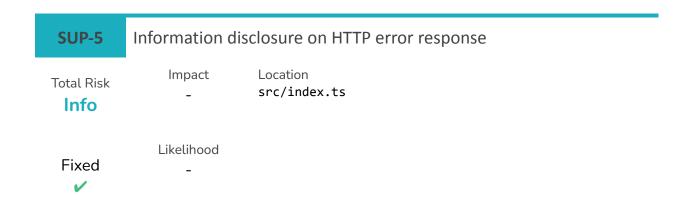
Recommendation

Add the necessary validations to make sure transactions are effectively broadcasted (both Bitcoin and Stacks transactions).

Status

Fixed in the reviewed version with commit c721f728401f266a738d88d1a233d79798b77038 from the feat/develop branch. The error handling for broadcasted transaction checks for two specific errors, which both happen if the UTXO was already spent. This happens if the worker is re-running a redeem job (perhaps after a full restart of the worker context), or if the HTLC was

lready recov nis job.	vered after ex	piration. Any	other errors	are thrown, w	hich causes re	tries of



The HTTP server returns an internal error message which might reveal sensitive information about the web server execution stack to attackers. The following code snippet shows the err.message variable being returned in the HTTP response.

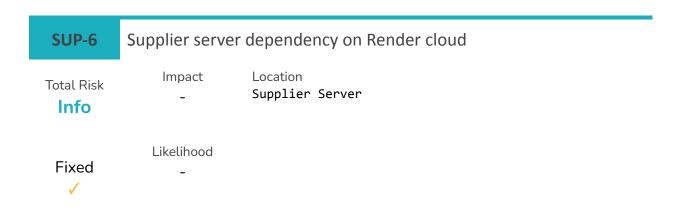
```
server.setErrorHandler((err, req, reply) => {
  logger.error(err);
  if (err instanceof Error) {
    console.error(err.stack);
    void reply.status(500).send({ error: err.message });
    return;
  }
  void reply.status(500).send({ status: 'error' });
  return;
});
```

Recommendation

Return a generic error message despite the error type generated.

Status

Fixed in the reviewed version with commit c721f728401f266a738d88d1a233d79798b77038 from the feat/develop branch.



The supplier server project is currently configured to work with Render cloud services only, making Render a direct dependency of the Magic Protocol bridge. In the event of a contingency in Render operation, liquidity suppliers could lose funds.

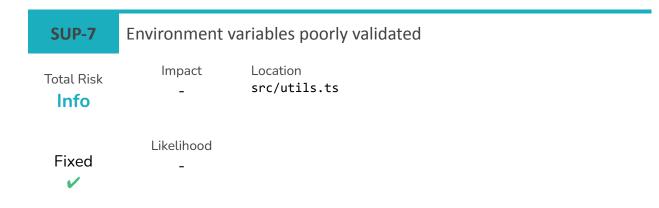
The project lacks configurations to persist Redis *in-memory* information to disk when not hosted on Render. A Redis database hosted outside of Render that is reset or crashes would lose the in-memory registries, leading to a reprocess of the contract events once the database reboots and a resulting loss of funds.

Recommendation

Add configurations to ensure the persistence of in-memory registries to disk on Redis instances not hosted in the Render cloud.

Status

Partially fixed. A work-in-progress PR is drafted to support similarly easy deployments on fly.io and documentation (regarding ad-hoc deployment) will include a note that Redis persistence is important.



Environment variables are not sufficiently validated before processing bridge events, specifically SUPPLIER_BTC_KEY and SUPPLIER_STX_KEY. If only one of these variables is incorrectly set, it can lead to inconsistencies in the behavior of the supplier server.

Below is a snippet of the SUPPLIER_BTC_KEY parsing in the getCompressedKey function. It is not clear why keys whose length is different than 66 characters are returned with isCompressed: true.

```
export function getCompressedKey(key: string) {
  if (key.length === 66) {
    const compressed = key.slice(64);
    return {
      key: key.slice(0, 64),
      isCompressed: compressed === '01',
    };
  }
  return { key, isCompressed: true };
}
```

Recommendation

Verify that the environment variables comply with expected formats and lengths. Validate all variables before starting bridge event processing to avoid inconsistencies.

Status

Fixed in the reviewed version with commit c721f728401f266a738d88d1a233d79798b77038 from the feat/develop branch. The worker thread now also runs a check to see if keys are configured properly when

starting. This checks on the on-chain supplier registry to ensure that the local environment config matches the registered supplier.



Bitcoin transactions use a constant fee which could be abused under certain network circumstances. If these fees turn out to be less than what's currently being used for the network, supplier servers would not be able to redeem BTC from HTLC transactions during the timeframe allowed.

The code snippet below shows the static fees used when finalizing the HTLC input in the redeem function:

```
const weight = 312;
const feeRate = 1;
const fee = weight * feeRate;
```

On the other hand, using too high fees would result in an unnecessary Bitcoin expenditure on miner fees, minimizing suppliers' earnings.

The issue is marked as Info as there is a T0D0 comment on the project about it. However, funds may be at risk.

Recommendation

Use a dynamic fee approach on Bitcoin transactions.

Status

Fixed in the reviewed version with commit c721f728401f266a738d88d1a233d79798b77038 from the feat/develop branch. The supplier now calls the estimatefee function from its configured Electrum server to get the appropriate fee rate. At the moment, it's aggressive in how many blocks it targets confirmations for, which could cause high fees (and potentially unprofitability)

during times of high congestion. Research needs to be done to either make this configurable (i.e. only target confirmations in X blocks) or to set a ceiling on the fee (which can lead to the issue documented here).

5. Disclaimer

The information presented in this document is provided "as is" and without warranty. Source code reviews are a "point in time" analysis, and as such, it is possible that something in the code could have changed since the tasks reflected in this report were executed. This report should not be considered a perfect representation of the risks threatening the analyzed system.