Belt Crosschain AMM Audit

Smart Contract Security Assessment

Nov 8, 2021

ABSTRACT

Dedaub was commissioned to perform a security audit on the Belt Finance Crosschain AMM contracts. We received the code base in the form of an archive file on Nov. 1, 2021. The primary contracts (non-test, non-interface, non-mock) are listed below:

```
crosschain/SwapReceiver.sol
crosschain/Router.sol
crosschain/exchanges/PCSExchangeProxy.sol
crosschain/exchanges/UniswapExchangeProxy.sol
crosschain/exchanges/KLAYswapExchangeProxy.sol
crosschain/exchanges/ExchangeProxy.sol
crosschain/exchanges/MDEXExchangeProxy.sol
stableSwap/Swap.sol
stableSwap/AmplificationUtils.sol
stableSwap/SwapDeployer.sol
stableSwap/saddleSwapUtils.sol
stableSwap/LPToken.sol
stableSwap/test/TestSwapReturnValues.sol
stableSwap/test/TestMathUtils.sol
stableSwap/SwapUtils.sol
utils/0.5/MathUtils.sol
utils/0.5/OwnerPausableUpgradeable.sol
utils/0.5/Proxy.sol
utils/0.5/GenericERC20.sol
utils/0.5/Context.sol
utils/0.5/SafeMath.sol
utils/0.5/Clones.sol
utils/0.5/ERC20BurnableUpgradeable.sol
utils/0.5/ERC20.sol
utils/0.5/Upgradeable.sol
utils/0.5/SafeERC20.sol
utils/0.5/RouterEncoder.sol
utils/0.5/Address.sol
utils/0.6/ReentrancyGuard.sol
utils/0.6/Ownable.sol
utils/0.6/Context.sol
utils/0.6/SafeMath.sol
utils/0.6/Address.sol
```

We mostly emphasized the contracts comprising the core crosschain functionality (`crosschain/*`), since StableSwap-related code was not part of the original audit scope

(did not exist during initial scoping), and is essentially a clone of well-audited code of Saddle (https://github.com/saddle-finance/saddle-contract). The `utils` folder primarily consists of Open Zeppelin contracts.

The audit also examined any other functionality highly related to the included contracts. More specifically, Orbit Bridge and the Belt-supported exchange services contracts.

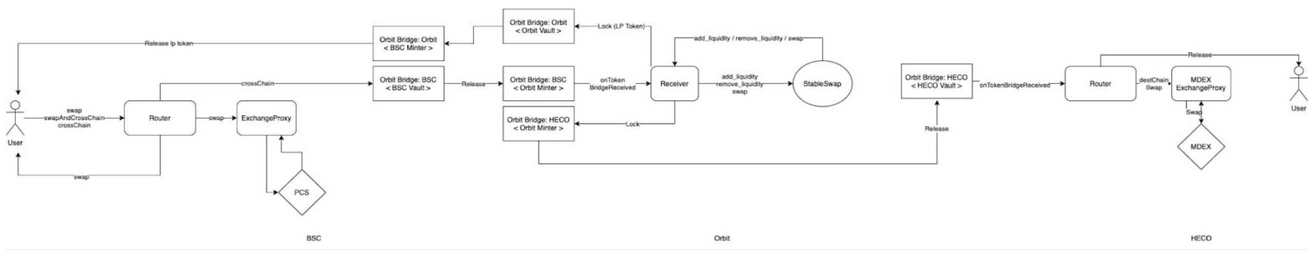
SETTING & CAVEATS

The exhaustively audited code base is of rather small size, at nearly 1KLoC but of considerable complexity. The audit's main target is security threats, i.e., what the community understanding would likely call "hacking", rather than regular use of the protocol. Functional correctness (i.e. issues in "regular use") is a secondary consideration. Functional correctness relative to low-level calculations (including units, scaling, quantities returned from external protocols) is generally most effectively done through thorough testing rather than human auditing.

ARCHITECTURE & HIGH-LEVEL RECOMMENDATIONS

The system provides a crosschain AMM service for stablecoins based on the StableSwap protocol first implemented and established by Curve for the Ethereum blockchain. Belt uses Saddle's implementation of the protocol in Solidity. Belt's StableSwap is deployed in the Orbit Chain. Additionally, the Orbit Bridge crosschain mechanism is utilized for the transaction-related crosschain message-passing from any of the supported chains to Orbit Chain and vice versa. OrbitBridge essentially offers the fundamental infrastructure via the off-chain operators and validators functionality responsible for the message passing and verification, along with Minters and Vaults contracts for token minting/burning and locking, respectively.

Belt's crosschain AMM basic structure is represented in the following diagram, provided by the developers.


Release lp token
Orbit Bridge: Orbit < BSC Minter >
Orbit Bridge: Orbit < Orbit Vault >
Lock (LP Token)
add_liquidity / remove_liquidity / swap
crossChain
Orbit Bridge: BSC < BSC Vault >
Release
Orbit Bridge: BSC < Orbit Minter >
onToken BridgeReceived
Receiver
add_liquidity remove_liquidity swap
StableSwap
Orbit Bridge: HECO < HECO Vault >
onTokenBridgeReceived
Router
destChain Swap
MDEX ExchangeProxy
Release
User
swap swapAndCrossChain crossChain
Router
swap
ExchangeProxy
Orbit Bridge: HECO < Orbit Minter >
Lock
Release
Swap
MDEX
User
swap
PCS
BSC
Orbit
HECO

One of the two main contracts, Router, is responsible for:

a) receiving users' requests for crosschain swaps (or liquidity providing) and forwarding them to the Orbit Chain where the core AMM functionality resides,

b) finalizing a crosschain swap request by unlocking the swapped amount from the Orbit Vault of the corresponding chain of interest and releasing the tokens to the final receiver.

A second Belt contract, SwapReceiver, plays a crucial role and is deployed in the Orbit Chain. SwapReceiver essentially lies between the two Routers' responsibilities and manages the StableSwap requests. SwapReceiver is responsible for giving back the user's deposited amount in case that the requested swap fails to complete. Note that any fees needed for such a (crosschain) refund to take place are paid by Belt. A DoS attack is thus possible as described in H2. Also note that apart from the stablecoin crosschain exchange service itself, Belt also supports "local" swaps through proxy contracts of well known exchange services, in the chain of the requester or the receiver, meaning either before or after the StableSwap. Belt supports only specific exchange services, however these are not somehow whitelisted resulting in an attack vector described in H1.

Overall, the codebase is well written and of professional standard. Apart from these High Severity issues the system's security highly relies on Orbit's crosschain functionality security, the largest part of which pertains to offchain activity and could not be audited. Lastly, we also make some recommendations that we believe could make the codebase clearer and easier to maintain.

VULNERABILITIES & FUNCTIONAL ISSUES

This section details issues that affect the functionality of the contract. Dedaub generally categorizes issues according to the following severities, but may also take other considerations into account such as impact or difficulty in exploitation:

Category	Description
CRITICAL	Can be profitably exploited by any knowledgeable third party attacker to drain a portion of the system's or users' funds OR the contract does not function as intended and severe loss of funds may result.
HIGH	Third party attackers or faulty functionality may block the system or cause the system or users to lose funds. Important system invariants can be violated.
MEDIUM	Examples: 01) User or system funds can be lost when third party systems misbehave. 02) DoS, under specific conditions. 03) Part of the functionality becomes unusable due to programming error.
LOW	Examples: 01) Breaking important system invariants, but without apparent consequences. 02) Buggy functionality for trusted users where a workaround exists. 03) Security issues which may manifest when the system evolves.

Issue resolution includes "dismissed", by the client, or "resolved", per the auditors.

CRITICAL SEVERITY

[No critical severity issues]

HIGH SEVERITY:

ID	Description	STATUS
H1	`request.destChainSwapData.exchange` is chosen by the user	**RESOLVED**

The `request.destChainSwapData.exchange,` used to perform a "local" swap at the end of a cross-chain swap, is controlled by the user and could be set to an arbitrary malicious contract. Such a malicious exchange could be used to steal any token potentially owned by the Router (not just tokens transferred to the Router by the user during a swap).

Such an attack is also facilitated by the following issues in the Router contract.

1. In `_swap`, an approval is given to the exchange in order to do the swap, but it is not removed afterwards, and could potentially remain valid at the end of the call.
2. In `_swap`, the code `succeed = amountOut >= request.amountOut` ignores the original `succeed` status, the Router will think that a swap failed even if the `address(proxy).call` transaction did not revert.
3. `onTokenBridgeReceived` has no reentrancy guard, allowing a reentrancy of the form `onTokenBridgeReceived-> _swap -> proxy.call -> swap -> _swap`

The above issues allow the following three attacks (which are related but substantially different in their logic), used to steal any token owned by the Router.

1. `request.destChainSwapData.exchange` is set to a malicious contract which simply steals the tokens to be swapped, and sends nothing back to the Router (without reverting). The Router, due to the `succeed = amountOut >= request.amountOut` code, believes that the swap failed, so it returns the tokens to the user, so the user receives the tokens twice.
2. `request.destChainSwapData.exchange` is set to a malicious contract which simply reverts. The Router, since the swap failed, returns the token to the user, but the user can collect the tokens a second time via the approval that the malicious contract still retains for the tokens.
3. Using `Router.swap` an adversary can perform a "pseudoTransfer", meaning a transfer of tokens to the router while retaining an approval on the transferred tokens (this is possible using again a malicious exchange, and the fact that the approval remains valid after the swap, if the exchange does not consume it). A pseudoTransfer increases the Router's balance artificially, making it appear that the Router has more tokens, while the adversary can reclaim them at any time via his approval.

Such pseudoTransfer can be used in an attack as follows: `destChainSwapData.exchange` is set to a malicious contract which first performs a normal swap. Then, it reenters in Router.swap, and performs a "pseudoTransfer", virtually increasing the Router's balance for the swap's target token. After the exchange finishes, `amountOut` is arbitrarily large (because of the pseudoTransfer), which leads to the Router returning more tokens than it should.

The severity of the above attacks is reduced by the fact that the Router does not currently seem to hold any tokens by design (so there's nothing to steal, except for tokens that the Router received as extraneous fee or by accident). Still, the Router could potentially hold tokens in some future update, and the attack is very broad, allowing to steal any amount from any token owned by the Router.

To fix this vulnerability and strengthen security checks, we suggest:

1. Limiting `request.destChainSwapData.exchange` to a whitelist of pre-approved exchange proxies.
2. To zero the approval to the exchange proxy after the swap.
3. To have `succeed = false` only if the exchange reverted, in which case we are sure that the tokens to be swapped are left intact (no need to check the swapped amount, both the exchange proxy and the exchange itself should already perform such a check).
4. Add `nonReentrant` to `Router.onTokenBridgeReceived`.

H2	Fee-based DoS	**RESOLVED**

In the current architecture, there are two types of Orbit fees:

- **User-side:** Orbit fee (or tax) to transfer the assets from the source chain to the Orbit chain where the swap takes place. These are paid by the user.
- **Belt-side:** Orbit fee to transfer the swapped assets to the target chain, or send the assets back to the source chain (`_giveBack`), in case of swap failure. These are paid by Belt without explicitly collecting any funds from the user to cover them.

As a consequence, a malicious user can force Belt to repeatedly pay the Belt-side fees by sending "dummy" swap requests. These can either be swaps that fail (with small amountOut), causing a _giveBack, or even swaps that succeed for small amounts. Depending on the amount of the User-side fees, a malicious user might have incentives to perform such an attack. Note that there are source chains without any fee (e.g., BSC), which facilitates such an attack. Note also that an Orbit tax alone (a percentage of the transferred amount) cannot prevent this attack, since the user is free to ask for swaps of arbitrarily small amounts.

One possible solution would be to collect a fee from the user to cover for the Belt-side Orbit fees.

MEDIUM SEVERITY:

[No medium severity issues]

LOW SEVERITY:

[No low severity issues]

OTHER/ ADVISORY ISSUES:

This section details issues that are not thought to directly affect the functionality of the project, but we recommend addressing them.

ID	Description	STATUS
A1	Give back the tokens in case of _requestBridge failure without the need of an escrow	**DISMISSED**

In SwapReceiver::onTokenBridgeReceived, if _requestSwap fails, the tokens are sent back to the user (via _giveBack). However, if _requestSwap succeeds but _requestBridge fails, the tokens are not sent back (since they are already swapped). The comment in line 197 suggests that an escrow account will be used to store the funds in this case.

An idea to consider is: to avoid an escrow account, both the `_requestSwap` and `_requestBridge` operations can be performed **inside an internal transaction**. If `_requestBridge` fails, the transaction can be reverted, which will also **undo the swap**, restoring the source tokens. Then `_giveBack` can be simply used to send the source tokens back to the user (similarly to the case when the swap fails).

Note that this is just a functional suggestion, not a security issue.

A2	`Router::_requestBridge` `recipient` argument is best renamed	**DISMISSED**

The `recipient` argument of `Router::_requestBridge` should best be renamed to `receiver`, to match the distinction between the two concepts throughout the rest of the code.

A3	Unnecessary ABI encoding/decoding of TxRequest data	**RESOLVED**

In Router, there is some unnecessary ABI encoding/decoding of TxRequests:

```
function swapAndCrossChain(... TxRequest memory txRequest) ... {
   ...
   _crossChain(swapRequest.path[swapRequest.path.length - 1], amountOut,
               abi.encode(txRequest));
}

function crossChain(... TxRequest memory txRequest) ... {
   ...
   _crossChain(token, amount, abi.encode(txRequest));
}

function _crossChain(address token, uint amount, bytes memory txData) private {
   ...
   TxRequest memory txRequest = abi.decode(txData, (TxRequest));
   _requestBridge(token, txRequest.receiver, amount, "ORBIT",
                  abi.encode(txRequest));  // Dedaub:same as txData
   ...
}
```

Either the last encoding can be removed, or the earlier two, so that `_crossChain` takes a `TxRequest` and encodes it.

A4	Seems dangerous and unnecessary to allow any minter to initiate a `SwapReceiver::onTokenBridgeReceived`	**RESOLVED**

The code of SwapReceiver::onTokenBridgeReceived has the form:

```
function onTokenBridgeReceived(address _token, ...) ... {
   ...
   bool isValidSender = false;
   for (uint i = 0; i < minters.length; i ++) {
      isValidSender = isValidSender || minters[i] == msg.sender;
   }
   ...
```

It is not clear why any minter would be acceptable as the `msg.sender`. Isn't `minterByToken[_token]` the only valid one? If not, this should be documented. Even though minters are trusted and registered, "confused deputy" attacks through trusted but misused entities are a common threat pattern. Furthermore, if there really needs to be a search for the right minter, the loop could be optimized to:

```
function onTokenBridgeReceived(address _token, ...) ... {
   ...
   bool isValidSender = false;
   for (uint i = 0; i < minters.length; i ++) {
      if (minters[i] == msg.sender) {
         isValidSender = true;
         break;
      }
   }
   ...
```

Although this is just an advisory item (since we cannot currently see security implications), unless there is a strong reason for iterating over possible minters, we strongly recommend addressing it. It is exactly the kind of "loose privilege" that an attacker may try to exploit.

A5	Workflow descriptive comments	**DISMISSED**

The overall readability of the codebase could significantly benefit from comments, e.g. above crucial functions, that describe the intended functionality in respect to the overall service's workflow and/or invariants that are to be respected under normal circumstances.

A6	Compiler known issues	**INFO**

Most of the contracts were compiled with the Solidity compiler v.0.6.12 while some of them with v.0.5.0. Both versions have some known bugs at the time of writing. We have inspected the list of bugs and believe that they do not affect the audited contracts.

DISCLAIMER

The audited contracts have been analyzed using automated techniques and extensive human inspection in accordance with state-of-the-art practices as of the date of this report. The audit makes no statements or warranties on the security of the code. On its own, it cannot be considered a sufficient assessment of the correctness of the contract. While we have conducted an analysis to the best of our ability, it is our recommendation for high-value contracts to commission several independent audits, as well as a public bug bounty program.

ABOUT DEDAUB

Dedaub offers technology and auditing services for smart contract security. The founders, Neville Grech and Yannis Smaragdakis, are top researchers in program analysis. Dedaub's smart contract technology is demonstrated in the contract-library.com service, which decompiles and performs security analyses on the full Ethereum blockchain.