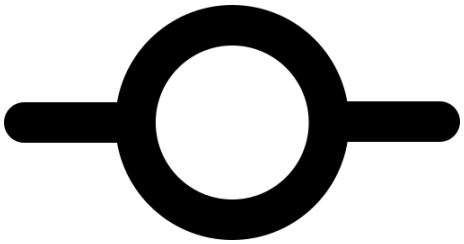
Counterstake Audit

Smart Contract Security Assessment

Nov. 17, 2021
(updated Mar. 4, 2022)



ABSTRACT

Dedaub was commissioned to perform a security audit on Counterstake's Decentralized Cross-chain Bridge Protocol smart contracts at commit hash `2aa1275ed41d58219226253922ba36b59bb0658b`. The code can be found in this Github repository.

SETTING & CAVEATS

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. Typically it can only be covered if we are provided with unambiguous (i.e., full-detail) specifications of what is the expected, correct behavior. In terms of functional correctness, we often trusted the code's calculations and interactions, in the absence of any other specification. 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. The financial model and incentives for participation (e.g., bonding curves, for favoring early investors) are taken for granted, although protocol-level financial manipulation (e.g., actions that will affect the price of participating in an Assistant pool) has been considered extensively.

The scope of this audit included the contracts comprising the Counterstake decentralized protocol for cross-chain transfers as well as the implementation of the assistant contracts, aiming to speed up transfers for end-users by taking care of the interaction with the protocol for a given transfer and getting a reward in return.

The core functionality of the Counterstake protocol is exported using the Import.sol and Export.sol contracts, both of which inherit from the Counterstake.sol contract and make use of the CounterStakeLibrary.sol library. Both the Import and Export contracts make use of the same governance functionality using the Governance.sol and VotedValue*.sol contracts in order to decide on the new values for a set of parameters affecting the contracts in a decentralized manner.

In terms of functionality, the Export contract is used to export an asset out of its native chain using the `transferToForeignChain(string memory foreign_address,`

`string memory data, uint amount, int reward)` method, as well as start a new claim for a repatriation of exported tokens from the foreign chain, getting the native tokens back. The Import contract is deployed on the foreign chain and is used to initiate a claim for the exported tokens. If a claim is successful, new foreign tokens are minted and sent to the recipient address. Additionally, users can use the `transferToHomeChain(string memory home_address, string memory data, uint amount, uint reward)` of the Import contract to burn the foreign tokens and return them to their native chain.

The claim functionality works with the claimant providing a stake worth more (currently 150%) than the claimed amount. For claims at the Export contract the stake token is the same as the native/exported token but for the Import contract this has to be a different token, creating a need for a price oracle in order to determine the adequate stake amount. The current price oracle is a centralized one, allowing the oracle's owner to set the relative prices of different assets. Because of the system's reliance on that oracle we suggest the use of widely-adopted decentralized oracle systems.

After a claim is initiated in the Import or Export contracts it initiates a challenge period where other participants of the Counterstake protocol can collectively pool their stake tokens against the claim, succeeding if they meet a specific amount higher than the claim. If the challenge is successful, another round in support of the claim begins, with an increased stake token amount. This process is repeated until a decision is reached, with the supporters of the winning side sharing the stakes of the losing side, proportionally to their contribution to the win.

When a transfer is initiated, the reward parameter is used to indicate whether the sender wants an assistant to aid with the cross-chain transfer, sending them the transfer amount minus the reward and taking over the claim process. There are currently 2 assistant contracts, ImportAssistant.sol and ExportAssistant.sol, each controlled by a manager who decides which claims and challenges the assistant pool will participate in. ExportAssistant is the simpler one, with the users pooling their native/to-be-exported tokens. The ImportAssistant keeps both the stake and the exported asset and uses an AMM style constant product formula to denote the relative prices of the two assets (also supports swapping between the two assets).

In addition to the issues of the two Assistant contracts (see M1, M2) in their current implementations, the rewards a user gets when withdrawing their shares are proportional to the balance the contract holds at the time (not considering the amount currently in claims and challenges). This means that a user should prefer to withdraw when the assistant pool is not utilized, while it should be expected that an assistant pool should participate in many claims and challenges to maximize its profit. *This can create a situation where a manager chooses to participate in many claims, trapping the liquidity providers of an assistant pool that want to get an adequate reward for their shares.*

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:

[No high severity issues]

MEDIUM SEVERITY:

ID	Description	STATUS
M1	Assistant withdrawal susceptible to sandwich/MEV attacks	**RESOLVED**[1]
Assistant Claim withdrawals are susceptible to sandwich/MEV attacks. More specifically: 1. Attacker sees large withdrawal, that will yield a high stake asset profit to the assistant contract (after a series of challenges). 2. Buys large amount of shares. The staked amounts are accounted for in the share price, but profits are not. 3. Assistant withdrawal tx gets executed, Assistant contract now holds a large amount of stake asset. 4. Assistant shares are now worth more - attacker withdraws his shares and makes profit. Using a more pessimistic strategy in the assistant share purchase logic which accounts for future rewards should help prevent such scenarios.		
M2	Swapping API can lead to frontrunning and other issues	**RESOLVED**
The API of the swapping/AMM functionality (`swapImage2Stake`/`swapStake2Image`) of ImportAssistant does not specify a minimum number of output tokens. Due to this, swap operations are susceptible to sandwich/MEV attacks, and/or unexpected slippage. We suggest the addition of a `minAmountOut` argument to the swap functions to prevent such issues.		

[1] Issue resolved by diffusing profits over time. This approach should be effective, assuming sane and appropriate parameterization

LOW SEVERITY:

ID	Description	STATUS
L1	Lack of support for non-standard ERC20 tokens	**RESOLVED**

The current implementation of the protocol and Assistant contracts does not support tokens deviating from the ERC20 standard by not returning a boolean success value using returndata (USDT is the most popular such token). We suggest the use of the OZ SafeERC20 wrapper library to support both standard and non-standard token implementations.

ID	Description	STATUS
L2	Assistant contracts can be bricked by manager misconfiguration	**RESOLVED**

The current implementation of the Assistant contracts allows the manager to pass their role to a new user using the `assignNewManager()` function. The function currently allows the new manager value to be 0, as also indicated by the comment above its definition:

```
// zero address is allowed
function assignNewManager(address newManager) onlyManager external {
   emit NewManager(managerAddress, newManager);
   managerAddress = newManager;
}
```

However, doing so would effectively brick the Assistant contract as there is no way to recover from it.

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	Immutable state variables in Factories	**RESOLVED**

All `*Factory` state variables are only set during contract construction. It is recommended that they be declared as immutable for gas saving.

A2	Explicit access modifier of state variables	**RESOLVED**

In `AssistantFactory`, state variables `governanceFactory` and `votedValueFactory` are implicitly private. It is recommended that they be explicitly declared with the `private` modifier to improve readability and be consistent with the rest of the codebase.

A3	AssistantFactory master contract naming convention	**RESOLVED**

In most `*Factory` contracts, the master copy contracts are stored in fields that follow the naming scheme `<contract name>Master`, however this is not the case with `AssistantFactory`, where the master contract state variables are named `exportAssistantFactory` and `importAssistantFactory`.

It is recommended that they be renamed, as to follow the above naming scheme, as it makes the code more consistent and clear.

A4	Commented code segments	**OPEN**

Throughout the codebase there are some commented out segments of code. It is good practice to remove such artifacts from the final-deployed code.

A5	Comments discussing reentrancy in ExportAssistant are wrong	**RESOLVED**

In the ExportAssistant contract above the implementations of the claim() and challenge() functions the comments note that "*reentrancy is probably not a risk unless a malicious token makes a reentrant call from its balanceOf, so nonReentrant can be removed to save 10K gas*". Indeed a contract cannot reenter using its balanceOf() function because (given that the caller has a correct interface) it is called using STATICCALL. However, after the control-flow is passed to the Export contract the token's state-altering transferFrom/transfer functions are called, allowing a malicious token to potentially reenter. Due to this we believe the nonReentrant modifiers should be kept in place, if the possibility of a malicious token cannot be otherwise precluded.

A6	Code reuse	OPEN

Throughout the contracts, the code segments to receive or send ETH/ERC20 tokens are reused in many places. We suggest moving this functionality to internal functions to be used in various places.

A7	Governance functionality ExportAssistant is unused	RESOLVED

The current implementation of the ExportAssistant contract has a governance module that serves no purpose. It can be removed to reduce the gas cost of its deployment.

A8	Initializers do not respect validators	LARGELY RESOLVED

Each contract that has state variables which can be set via governance voting has some validation functions which ensure that the proposed values are safe/within spec. However, many initializer functions which set the same state variables, do not validate the provided values, which may lead to some inconsistencies. For example:

```
contract Import is ERC20, Counterstake {
    function initImport(...,  address oracleAddr) public {
        // [...]
        // Dedaub: No validation of oracleAddr
        oracleAddress = oracleAddr;
        // [...]
    }

    function validateOracle(address oracleAddr) view external {
        require(CounterstakeLibrary.isContract(oracleAddr), "bad oracle");
        (uint num, uint den) = getOraclePrice(oracleAddr);
        require(num > 0 || den > 0, "no price from oracle");
    }

    // [...]
}
```

It is recommended that all initializers validate their input, to prevent human error and be consistent with the rest of the code.

A9	Potential problems with the format of claimId	OPEN

The claimId of a given claim is computed by the following code segment:

```
function getClaimId(
         string memory sender_address, address recipient_address,
         string memory txid, uint32 txts, uint amount, int reward,
         string memory data) public pure returns (string memory){
   return string(abi.encodePacked(sender_address, '_',
          toAsciiString(recipient_address), '_', txid, '_',
          uint2str(txts), '_', uint2str(amount), '_',
          int2str(reward), '_', data));
}
```

While generally using more than one variable length array together with `abi.encodePacked()` is considered a bad smell, we do not believe it can lead to any issues in this case because for every deployment of the protocol's contracts `sender_address` and `txid` should have a constant expected length. Furthermore, the separator ('_') should not appear in the data, except in marker values.

In addition, the use of the string type for the parameters of a claim can cause confusion between users (two different assistants initiating claims for the same transfer using different capitalizations for the same hexadecimal address or txid). We instead suggest the use of the bytes type for these parameters which will remove such issues and also enable the removal of the low-level code that deals with string conversions.

A10	Floating pragma	**OPEN**

The floating pragma `pragma solidity ^0.8.3;` is used in most contracts, allowing them to be compiled with any of the 0.8.3 - 0.8.10 versions of the Solidity compiler. Although the differences between these versions are small, floating pragmas should be avoided and the pragma should be fixed to the version that was used for the contracts' deployment (Solidity version 0.8.6 according to the deployed contract info on Etherscan).

A11	Compiler known issues	**INFO**

The contracts have been compiled and deployed with the Solidity compiler v0.8.6 which, at the time of writing, has some known bugs. We inspected the bugs listed for version 0.8.6 and believe that the subject code is unaffected.

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.