Dedaub

Security Technology for Smart Contracts

# DRC DigitalReserve V1.0

## Smart Contract Security Assessment

Date: Feb. 10, 2021

# Abstract

Dedaub was commissioned to perform a security audit on DRC's digital reserve smart contracts. The digital reserve is a vehicle for investing DRC's token (not included in this audit, but is fairly straightforward). The digital reserve is in turn backed by several other tokens traded using the UniswapV2 protocol. The underlying currencies of the digital reserve and proportions thereof are expected to be set and maintained by a trusted entity (e.g., a fund manager) working in the interest of the DRC token holders. The digital reserve itself is a fungible ERC-20 token (DR-POD).

Four auditors worked on the task over the course of three working days. We reviewed the code in significant depth, assessed the economics of the protocol and processed it through automated tools. We also decompiled the code and analyzed it, using our static analysis (incl. symbolic execution) tools, to detect possible issues.

# Setting and Caveats

The code base is relatively small in size, however the economic mechanisms behind the digital reserve are complicated. Although most smart contract auditors do not account for protocol composability issues and the economic risks these bring, we do in this audit.

The audit focused on security, establishing the overall security model and its robustness and also crypto-economic issues. Functional correctness (e.g., that the calculations are correct) was a secondary priority. Functional correctness for this project can be assessed through more thorough testing.

## Trust Model/Centralization Elements

[*This section is included for context, although its contents should already be known to the commissioner of an audit.*]
The contract cannot directly acquire the user's funds. The funds remain with the caller until explicitly converted. However, the owner of the Digital Reserve contract can change the allocation strategy at any point, including to otherwise value-less tokens. In this sense, the contract's users are fully trusting the contract owner to promote the best interests of the pooled funds.

Full list of other audit findings can be found below.

## Critical Severity

<table>
<tr><th>Description</th><th>Status</th></tr>
<tr><td>
The digital reserve is susceptible to flash loan attacks. Hackers can potentially “leech” all the underlying tokens invested in the reserve. Obviously, such a flash loan attack would require certain conditions to be met (e.g., favorable sizes of liquidity pools) and a skilled hacker to pull off but it can be done.<br><br>
We will consider much simplified numbers in order to demonstrate the attack and the weak points of the economic model.<br><br>
Imagine that we start from the following state in the Vault and UniswapV2 pools respectively:<br><br>
<table>
<tr><th>Vault (DigitalReserve)</th><th>Pool (UniswapV2 WETH ↔ WBTC pool)</th></tr>
<tr><td>5 DRC deposited, the strategy is very simple: all invested in a single strategy token (WBTC), assume 5 WBTC in total<br><br>1 POD minted (not realistic, just for illustration, one could use any number and all the other POD numbers would scale accordingly)</td><td>**10 WBTC ↔ 200 WETH**</td></tr>
</table><br>
At this point the real exchange rate is **1 DRC ↔ 1 WBTC ↔ 20 WETH**. That’s the actual fair value. But we’ll manipulate the Uniswap pool during the transaction, so this will change. Updates to the balance of WBTC ↔ WETH in the pool will be shown in bold and highlighted. [Attacker is a “he” in the explanation below, for directness.]
</td><td>**Resolved**</td></tr>
</table>

**Step 1.** The attacker dumps 10 WBTC, to lower the Uniswap WBTC price. Get back 100 ETH, based on Uniswap's constant-product algorithm. Pool is now **20 WBTC ↔ 100 WETH.**

At this point the attacker has **lost** money in order to distort the value of the Uniswap pool. Whoever next trades WETH for WBTC will be making back the money. (It will be the attacker himself, but he will do it through the vault, and this will give him double benefit.)

**Step 2.** The attacker deposits to the vault (DigitalReserve) 5 DRC. Its fair market value is 100 ETH. The Uniswap pool DRC ↔ WETH is not manipulated, so we don't show it: it will remain fairly priced at 100 ETH.

The vault has 5 DRC from past deposits, stored as 5 WBTC. So when Uniswap is asked (`currentPodUnitPrice` in `depositDrc`() indirectly calls Uniswap's `getAmountsOut`) it says that the vault (which is 1 POD) is worth just 20 ETH. (It's really worth 100 ETH.)

*[The Uniswap getAmountsOut computes based on the formula* `amountOut = amountIn * reserveOut / (reserveIn + amountIn)`*. It also adds 0.3% fees, which we ignore for simplicity. In this case we are asking how much we'd get out if we were to trade amountIn =5 WBTC, with the reserves being 20 WBTC ↔ 100 WETH.]*

The attacker's 100 ETH (from the 5 DRC deposited) is exchanged into 10 WBTC. The pool now is **10 WBTC ↔ 200 WETH.** The pool is back to "fair". The attacker made back his losses, but also: this is money traded through the vault. So the vault now has 5+10 = 15 WBTC. The deposit function calls again the Uniswap `getAmountsOut`, which tells it that the Vault's current ETH value is 120 ETH. (This is what we would be getting if we were trading the entire vault in the Uniswap pool.) [15 * 200 / (15 + 10) = 120]

The computation in the smart contract is next determining how much the attacker's deposit added to the value of the vault. That's 120 ETH / 20 ETH (previously computed price of a POD) = 6 POD. So, the attacker's deposit increased the vault's assets from 1 POD to 6. The code mints the attacker 5 POD.

| | |
|---|---|
| To recap, the attacker spent 10 WBTC + 5 DRC (fair value = 300 ETH) but ended up with 350 ETH: 100 ETH from the swap of Step 1 and 250 ETH from 5 POD (out of a total of 6 POD, which all together map to 15 WBTC, or 300 ETH).<br>The attacker made money because when he deposited 5 DRC to get POD, the vault bought for him WBTC, making back slippage losses from the first step. But at the same time, the vault valued its current assets by considering a swap from WBTC to WETH. But WBTC was way undervalued at the time. When the attacker gained back the slippage and the vault tried to estimate how much value the attacker added to it, this was fixed. Therefore, while making back his slippage losses from step 1, the attacker also cheats the vault to give him a higher percentage of the total POD than it should have given. | |

## High Severity

*[No high severity issues]*

## Medium Severity

*[No medium severity issues]*

## Low Severity

| Description | Status |
|---|---|
| If the number of strategy tokens increases to more than 255 a number of variables and also loop induction variables will overflow. | **Resolved** |

## Lowest/Style/Info/Suggestions

| Description | Status |
|---|---|
| Some fields could have stricter modifiers: | **Resolved** |

| | |
|---|---|
| ● `router` could become `immutable`<br>● `uniswapRouter` could become `immutable`<br>● `_pricerDecimals` can become `constant` | |
| There are some redundant fields:<br>● `router` and `uniswapRouter` point to the same contract.<br>● `_stategyTokenCount` is functionally dependent on `_stategyTokens.length`<br>It is suggested that only one field per pair is maintained. | **Resolved** |
| It is possible that `_stategyTokens` and `_tokenPercentage` be merged into a storage array of type:<br>`struct StategyToken {`<br>`    address tokenAddress;`<br>`    uint8 tokenPercentage;`<br>`}`<br>Since these storage structures are accessed in tandem (usually _tokenPercentage[_strategyTokens[i]]), this change should lead to significant gas savings as the two fields would be stored in a single storage word. | **Resolved** |
| `_convertEthToStrategyTokens()` has a return value that is never used at its call sites. | **Resolved** |
| Strictly speaking, the fees are 1/99, i.e., 1.01%, not 1%. Mentioning for information purposes, as it may be understood already. | **Resolved** |
| `changeStrategy()` performs token-to-ETH and ETH-to-token swaps even when these are unnecessary. This is probably fine, but it does have an impact on Uniswap fees. A future version can lower the fees, if this becomes an issue. However, since `rebalance()` already performs such calculations, it is not clear why `changeStrategy()` and `rebalance()` cannot be unified into a single generalized-rebalance routine. | Open |
| Some variables are not explicitly initialized:<br>● `proofOfDepositPrice` in `getProofOfDepositPrice()`<br>● `totalWorthInEth` in `rebalance()`<br>● `amountOut` in `_getEthAmountByStrategyTokensAmount()` | **Resolved** |

| | |
|---|---|
| ● `ethConverted` in `_convertStrategyTokensToEth()` | |
| Use of a floating pragma: The floating pragma `pragma solidity ^0.6.6;` is used allowing the contracts to be compiled with the `0.6.6 - 0.6.12` versions of the Solidity compiler. Although there differences between these versions are small, floating pragmas should be avoided and the pragma should be fixed to the version that will be used for the contracts' deployment. | **Resolved** |
| The contracts were compiled with the Solidity compiler `v0.6.12` which has some known minor issues (but relatively few, compared to earlier versions). We have reviewed the issues and do not believe them to affect the contract. More specifically, at the time of writing, there are 2 known compiler bugs associated with the Solidity compiler `v0.6.12:`<br>● Copying an empty `bytes` or `string` array from memory to storage can cause data corruption. (We couldn't find bytes arrays in storage.)<br>● Direct assignments of storage arrays with an element size <= 16 bytes (more than one values fit in one 32 byte word) are not correctly cleared if the length of the newly assigned value is smaller than the length of the previous one. (No such array is ever stored.) | **Closed** |

## 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 status 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.