Audit of ERC95

Audit of The CORE ERC95 Smart Contracts

a report of findings by

Van Cam Pham, PhD

innovative fortuna iuvat

November 5th, 2020

Table of Contents

Document Info

Client	cVault Finance
Title	Smart Contract Audit
Auditors	Van Cam Pham, PhD
Reviewed By	Joel Farris
Approved By	Rasikh Morani

Contact

For more information on this report, contact The Arcadia Media Group Inc.

Rasikh Morani
(972) 543-3886
rasikh@arcadiamgroup.com
https://t.me/thearcadiagroup

Executive Summary

A Representative Party of the cVault Finance ("cVault.Finance") engaged The Arcadia Group ("Arcadia"), a software development, research and security company, to conduct a review of the following ERC95 smart contracts on the CORE-v2 repo at Commit #c85f0e6ac308a02798337973576b97c4d14c26b6.

ERC95.sol
cBTC.sol

Arcadia completed the review using various methods primarily consisting of dynamic and static analysis. This process included a line by line analysis of the in-scope contracts, optimization analysis, analysis of key functionalities and limiters, and reference against intended functionality.

Findings

1. Initializer input parameters

- ERC95-1
- Severity: Medium
- Likelihood: Low
- Impact: Medium

- Target: ERC95.sol
- Category: Low
- Finding Type: Dynamic
- Lines: 85-138

In the ERC95 contract, the constructor takes `_percent` as its input.

- This array input parameter should be checked for its item values less than 100 percent as it cannot be greater than 100 per;
- Line 102, the sum of all array items in `_percent` should be computed using `SafeMath` to avoid overflow that can result in `_percent` item values greater than 100 but the sum of all items `percentTotal` is still 100;
- Not using `SafeMath` can result in invalid percentage values (greater than 100), which can cause the contract function incorrectly. Due to not using `SafeMath` for summing percentage values, the transaction that initiates the contract can be succeeded but having invalid percentage values, which can open doors for exploitation later.

```
 for (uint256 loop = 0; loop < _addresses.length; loop++) {
                // 0 % tokens cannnot be permitted
                require(_percent[loop] > 0 ,"ERC95 : All wrapped tokens have to have at least
1% of total");

                // we check the decimals of current token
                // decimals is not part of erc20 standard, and is safer to provide in the
caller
                // tokenDecimals[loop] = IERC20(_addresses[loop]).decimals();
                decimalsMax = tokenDecimals[loop] > decimalsMax ? tokenDecimals[loop] :
decimalsMax; // pick max

                percentTotal += _percent[loop]; // further for checking everything adds up
                //_numTokensWrapped++; // we might just assign this
                numTokensWrapped++;
                console.log("loop one loop count:", loop);
            }
```

Action Recommended:

- Check `_percent` array item values less than or equal to 100;
- Use SafeMath for computing the sum of `_percent`;

```
for (uint256 loop = 0; loop < _addresses.length; loop++) {
                // 0 % tokens cannnot be permitted
                require(_percent[loop] > 0 ,"ERC95 : All wrapped tokens have to have at least
1% of total");
                require(_percent[loop] <= 100 ,"ERC95 : All wrapped tokens have to have at most
100% of total");

                // we check the decimals of current token
                // decimals is not part of erc20 standard, and is safer to provide in the
caller
                // tokenDecimals[loop] = IERC20(_addresses[loop]).decimals();
                decimalsMax = tokenDecimals[loop] > decimalsMax ? tokenDecimals[loop] :
decimalsMax; // pick max

                percentTotal = percentTotal.add(_percent[loop]); // further for checking
everything adds up
                //_numTokensWrapped++; // we might just assign this
                numTokensWrapped++;
                console.log("loop one loop count:", loop);
            }
```

2. Redundant local variable `numTokensWrapped`

- ERC95-2
- Severity: Low
- Likelihood: Low
- Impact: Low

- Target: ERC95.sol
- Category: Informational
- Finding Type: Dynamic
- Lines: 92, 104, 109, 110

In the function `__ERC95_init` of the contract `ERC95`, the local variable `numTokensWrapped` is redundant. This is because if the contract initializer succeeds, the value of `numTokensWrapped` should be always `_addresses.length`. This leads to the redundancy of the statements at lines 104 and 109.

```
    uint8 numTokensWrapped = 0;
            for (uint256 loop = 0; loop < _addresses.length; loop++) {
                // 0 % tokens cannnot be permitted
```

```
                require(_percent[loop] > 0 ,"ERC95 : All wrapped tokens have to have at least 1% of
total");

                // we check the decimals of current token
                // decimals is not part of erc20 standard, and is safer to provide in the caller
                // tokenDecimals[loop] = IERC20(_addresses[loop]).decimals();
                decimalsMax = tokenDecimals[loop] > decimalsMax ? tokenDecimals[loop] : decimalsMax;
// pick max

                percentTotal += _percent[loop]; // further for checking everything adds up
                //_numTokensWrapped++; // we might just assign this
                numTokensWrapped++;
                console.log("loop one loop count:", loop);
            }

            require(percentTotal == 100, "ERC95 : Percent of all wrapped tokens should equal 100");
            require(numTokensWrapped == _addresses.length, "ERC95 : Length mismatch sanity check
fail"); // Is this sanity check needed? // No, but let's leave it anyway in case it becomes needed
later
            _numTokensWrapped = numTokensWrapped;
```

Action Recommended: For simplification, removing the local variable `numTokensWrapped` will be fine. This can be done by deleting lines 92, 104, and 109, while changing line 110 to:

```
            _numTokensWrapped = _addresses.length;
```

3. Missing checking `tokenDecimals`'s length

- ERC95-3
- Severity: low
- Likelihood: Low
- Impact: Low

- Target: ERC95.sol
- Category: Low
- Finding Type: Dynamic
- Lines: 89

In the ERC95 contract's initializer, the latter should verify that the number of token decimals in the input array `tokenDecimals` is equal to the length of array `_addresses`. This can save gas cost in case the length of `tokenDecimals` is less than that of `_addresses`.

Action Recommended: Add a `require` statement to check the length of `tokenDecimals` must be equal to the length of `_addresses`.

4. Deposit and Unwrap can fail with tokens having fees on transfer

- ERC95-4
- Severity: High
- Likelihood: Medium
- Impact: Medium
- Target: ERC95.sol
- Category: Medium
- Finding Type: Dynamic
- Lines: 175-182, 204-212, 236-262

In the ERC95 contract, any user can trigger a wrap action that can create a corresponding amount of the ERC95 token with respective to `_amountWrapperPerUnit`. If one of the supported tokens of the ERC95 token contract has fees on transfer, the function `sendUnderlyingTokens` can fail as follows:

- When depositing `_amt` of ERC95 token, function `_depositUnderlying` computes the corresponding amounts of wrapped tokens, for example wrapping `m` amount of token `X`.
- If `X` has fees or burn rate on transfer, the ERC95 token contract will receive an amount less than `m`.
- When the user decides to unwrap, in function `sendUnderlyingTokens`, the ERC95 token contract must send `m` amount of `X` to the user address. This transaction will fail because the ERC95 token contract has less than `m` token of `X`.

Due to this mismatch between the expected deposit and the actual deposit amount, the function `_updateReserves` will also fail due to an underflow at line 245. This is because the total actual deposit will be less than the reserve, which is the sum of all expected deposits.

```
  function sendUnderlyingTokens(address to, uint256 amt) internal {
          for (uint256 loop = 0; loop < _numTokensWrapped; loop++) {
              WrappedToken storage currentToken = _wrappedTokens[loop];
              uint256 amtToSend = amt.mul(currentToken._amountWrapperPerUnit);
              safeTransfer(currentToken._address, to, amtToSend);
              currentToken._reserve = currentToken._reserve.sub(amtToSend);
          }
      }

// Loops over all tokens in the wrap and deposits them with allowance
      function _depositUnderlying(uint256 amt) internal {
          for (uint256 loop = 0; loop < _numTokensWrapped; loop++) {
              WrappedToken memory currentToken = _wrappedTokens[loop];
              // req successful transfer
              uint256 amtToSend = amt.mul(currentToken._amountWrapperPerUnit);
              safeTransferFrom(currentToken._address, msg.sender, address(this), amtToSend);
              // Transfer went OK this means we can add this balance we just took.
              _wrappedTokens[loop]._reserve = currentToken._reserve.add(amtToSend);
          }
```

```
        }

function _updateReserves() internal returns (uint256 qtyOfNewTokens) {
            // Loop through all tokens wrapped, and find the maximum quantity of wrapped tokens that
can be created, given the balance delta for this block
            console.log("_numTokensWrapped: ", _numTokensWrapped);
            for (uint256 loop = 0; loop < _numTokensWrapped; loop++) {
                WrappedToken memory currentToken = _wrappedTokens[loop];
                uint256 currentTokenBal = IERC20(currentToken._address).balanceOf(address(this));
                console.log("currentTokenBal inside loop: ", currentTokenBal, currentToken._address);
                console.log("currentToken._amountWrapperPerUnit: ",
currentToken._amountWrapperPerUnit);
                // TODO: update to not use percentages
                uint256 amtCurrent =
currentTokenBal.sub(currentToken._reserve).div(currentToken._amountWrapperPerUnit); // math check pls
                console.log("amtCurrent: ", amtCurrent);
                qtyOfNewTokens = qtyOfNewTokens > amtCurrent ? amtCurrent : qtyOfNewTokens; // logic
check // pick lowest amount so dust attack doesn't work
                                                            // can't skim in txs or they have
non-deterministic gas price
                console.log("qtyOfNewTokens: ", qtyOfNewTokens);
                if(loop == 0) {
                    qtyOfNewTokens = amtCurrent;
                }
            }
            console.log("Lowest common denominator for token mint: ", qtyOfNewTokens);
            // second loop makes reserve numbers match from computed amount
            for (uint256 loop2 = 0; loop2 < _numTokensWrapped; loop2++) {
                WrappedToken memory currentToken = _wrappedTokens[loop2];

                uint256 amtDelta = qtyOfNewTokens.mul(currentToken._amountWrapperPerUnit);// math
check pls
                _wrappedTokens[loop2]._reserve = currentToken._reserve.add(amtDelta);// math check pls
            }
        }
```

Action Recommended:

- Because there is no standard for fees or burn on transfer tokens, it's hard, even impossible to check whether a token has fees or burns on transfer. This is because a token might support fees using a threshold-based, percentage-based, or flat fee mechanism. Thus, it is very hard to support those types of tokens. Therefore, before

deploying an ERC95 token, the deployer should verify that all supported tokens in the initializer should not have any fees or burns on transfer.

- A possible solution the team can consider is to recompute `_amountWrapperPerUnit` every time a new deposit is made. In a typical case where all supported tokens are no fees or burns on transfer, `_amountWrapperPerUnit` will be constant. In cases of tokens with fees or burns on transfer, `_amountWrapperPerUnit` will be adjusted with the fees or burns on transfers. The following is a suggested code portion for function `_depositUnderlying`.
- Because `_amountWrapperPerUnit` is adjusted every deposit (it's worth noting that the adjustment will be small because fees or burns on transfers are usually small), the user of `_reserve` is unnecessary. This method also improves on the aspect of excessive token quantity in function `skim`. The latter is only not useful for normal users as it will be called by any programmed bots written by developers to claim any excessive token quantity. By using the function `syncAmountPerUnit`, if there is any excessive token quantity, `_amountWrapperPerUnit` will be increased after any deposit or unwrap. The increase of `_amountWrapperPerUnit` will benefit all users that deposit tokens to the contract.

```
    function syncAmountPerUnit(uint256 tokenIndex, uint256 amt) internal {
        WrappedToken storage currentToken = _wrappedTokens[tokenIndex];
        uint256 currentTokenBal = IERC20(currentToken._address).balanceOf(address(this));
        uint256 supplyAfterMint = _totalSupply.add(amt);
        currentToken._amountWrapperPerUnit = currentTokenBal.div(supplyAfterMint);
    }

    // Loops over all tokens in the wrap and deposits them with allowance
    function _depositUnderlying(uint256 amt) internal {
        for (uint256 loop = 0; loop < _numTokensWrapped; loop++) {
            WrappedToken memory currentToken = _wrappedTokens[loop];
            // req successful transfer
            uint256 amtToSend = amt.mul(currentToken._amountWrapperPerUnit);
            safeTransferFrom(currentToken._address, msg.sender, address(this), amtToSend);
            syncAmountPerUnit(loop, amt);
        }
    }

    function sendUnderlyingTokens(address to, uint256 amt) internal {
        for (uint256 loop = 0; loop < _numTokensWrapped; loop++) {
            syncAmountPerUnit(loop, 0); //sync _amountWrapperPerUnit before unwrapping
            WrappedToken storage currentToken = _wrappedTokens[loop];
            uint256 amtToSend = amt.mul(currentToken._amountWrapperPerUnit);
            safeTransfer(currentToken._address, to, amtToSend);
        }
    }
```

Disclaimer

While best efforts and precautions have been taken in the preparation of this document, The Arcadia Group and the Authors assume no responsibility for errors, omissions, or for damages resulting from the use of the provided information. Additionally Arcadia would like to emphasize that use of Arcadia's services does not guarantee the security of a smart contract or set of smart contracts and does not guarantee against attacks. One audit on its own is not enough for a project to be considered secure; that categorization can only be earned through extensive peer review and battle testing over an extended period of time.