Yf-DAI TOKEN AUDIT

September 2020

BLOCKCHAIN CONSILIUM

Contents

Introduction

We first thank yfdai.finance for giving us the opportunity to audit their smart contract. This document outlines our methodology, audit details, and results.

yfdai.finance asked us to review their Yf-DAI token smart contract (ETH mainnet address: `0xf4CD3d3Fda8d7Fd6C5a500203e38640A70Bf9577`). Blockchain Consilium reviewed the system from a technical perspective looking for bugs, issues and vulnerabilities in their code base. The Audit is valid for `0xf4CD3d3Fda8d7Fd6C5a500203e38640A70Bf9577` Ethereum smart contract. The audit is not valid for any other versions of the smart contract. Read more below.

Audit Summary

This code is clean, thoughtfully written and in general well architected. The code conforms closely to the documentation and specification.

Overall, the code is clear on what it is supposed to do for each function. The visibility and state mutability of all the functions are clearly specified, and there are no confusions.

Audit Result	✓ **PASSED**
High Severity Issues	None
Moderate Severity Issues	None
Low Severity Issues	*4*
Improvement Suggestions	*2*

Overview

The project has one Solidity file for the Yf-DAI ERC20 Token Smart Contract, the YfDAIfinance.sol file that contains about 91 lines of Solidity code. We manually reviewed each line of code in the smart contract.

Methodology:

Blockchain Consilium manually reviewed the smart contract line-by-line, keeping in mind industry best practices and known attacks, looking for any potential issues and vulnerabilities, and areas where improvements are possible.

We also used automated tools like slither for analysis and reviewing the smart contract. The raw output of these tools is included in the Appendix. These tools often give false-positives, and any issues reported by them but not included in the issue list can be considered not valid.

Classification / Issue Types Definition:

1. **High Severity:** which presents a significant security vulnerability or failure of the contract across a range of scenarios, or which may result in loss of funds.
2. **Moderate Severity:** which affects the desired outcome of the contract execution or introduces a weakness that can be exploited. It may not result in loss of funds but breaks the functionality or produces unexpected behaviour.
3. **Low Severity:** which does not have a material impact on the contract execution and is likely to be subjective.

The smart contract is considered to pass the audit, as of the audit date, if no high severity or moderate severity issues are found.

Attacks & Issues considered while auditing

In order to check for the security of the contract, we reviewed each line of code in the smart contract considering several known Smart Contract Attacks & known issues.

- **Overflows and underflows:**
  An overflow happens when the limit of the type variable uint256 , 2 ** 256, is exceeded. What happens is that the value resets to zero instead of incrementing more.

  For instance, if we want to assign a value to a uint bigger than 2 ** 256 it will simple go to 0—this is dangerous.

  On the other hand, an underflow happens when you try to subtract 0 minus a number bigger than 0.For example, if you subtract 0 - 1 the result will be = 2 ** 256 instead of -1.

This is quite dangerous. This contract **DOES** check for overflows and underflows manually, but it also uses direct arithmetic operations, we highly recommend using **OpenZeppelin's** *SafeMath* for overflow and underflow protection.

- **Reentrancy Attack:**
  One of the major dangers of calling external contracts is that they can take over the control flow, and make changes to your data that the calling function wasn't expecting. This class of bug can take many forms, and both of the major bugs that led to the DAO's collapse were bugs of this sort.

  This smart contract does not include external calls and thus *is not vulnerable* to re-entrancy attack.

- **Replay attack:**
  The replay attack consists of making a transaction on one blockchain like the original Ethereum's blockchain and then repeating it on another blockchain like the Ethereum's classic blockchain. The ether is transferred like a normal transaction from a blockchain to another. Though it's no longer a problem because since the version 1.5.3 of *Geth* and 1.4.4 of *Parity* both implement the attack protection EIP 155 by Vitalik Buterin.
  So the people that will use the contract depend on their own ability to be updated with those programs to keep themselves secure.

- **Short address attack:**
  This attack affects ERC20 tokens, was discovered by the Golem team and consists of the following:

  A user creates an Ethereum wallet with a trailing 0, which is not hard because it's only a digit. For instance: `0xiofa8d97756as7df5sd8f75g8675ds8gsdg0`
  Then he buys tokens by removing the last zero:
  Buy 1000 tokens from account `0xiofa8d97756as7df5sd8f75g8675ds8gsdg`. If the contract has enough amount of tokens and the buy function doesn't check the length of the address of the sender, the Ethereum's virtual machine will just add zeroes to the transaction until the address is complete.

  The virtual machine will return 256000 for each 1000 tokens bought. This is abug of the virtual machine.

  Here is a **fix for short address attacks**

```
modifier onlyPayloadSize(uint size) {
    assert(msg.data.length >= size + 4);
    _;
}
function transfer(address _to, uint256 _value) onlyPayloadSize(2 * 32) {
```

```
    // do stuff
}
```

Whether or not it is appropriate for token contracts to mitigate the short-address attack is a contentious issue among smart-contract developers. Many, including those behind the OpenZeppelin project, have explicitly chosen not to do so. Blockchain Consilium doesn't consider short address attack an issue of the smart contract at the token level.

This contract **does not** implement an `onlyPayloadSize(uint numwords)` modifier for `transfer`, `transferFrom`, `approve`, `increaseAllowance`, and `decreaseAllowance` functions, it probably assumes that checks for short address attacks are handled at a higher layer (which generally are), and since the `onlyPayloadSize()` modifier started causing some bugs restricting the flexibility of the smart contracts, it's alright not to check for short address attacks at the Token Contract level to allow for some more flexibility for dAPP coding, but the checks for short address attacks must be done at some layer of coding (e.g. for buys and sells, the exchange can do it - almost all well-known exchanges check for short address attacks), this contract *does not prevent short address attack*, so the *checks for short address attack must be done while buying or selling or coding a DAPP using Yf-DAI where necessary*.

You can read more about the attack here: ERC20 Short Address Attacks.

- **Approval Double-spend:**
  Imagine that Parul approves Arun to spend 100 tokens. Later, Parul decides to approve Arun to spend 150 tokens instead. If Arun is monitoring pending transactions, then when he sees Parul's new approval he can attempt to quickly spend 100 tokens, racing to get his transaction mined in before Parul's new approval arrives. If his transaction beats Parul's, then he can spend another 150 tokens after Parul's transaction goes through.

  This issue is a consequence of the ERC20 standard, which specifies that `approve()` takes a replacement value but no prior value. Preventing the attack while complying with ERC20 involves some compromise: users should set the approval to zero, make sure Arun hasn't snuck in a spend, then set the new value. In general, this sort of attack is possible with functions which do not encode enough prior state; in this case Parul's baseline belief of Arun's outstanding spent token balance from the Arun allowance.

  It's possible for `approve()` to enforce this behaviour without API changes in the ERC20 specification:
  `if ((_value != 0) && (approved[msg.sender][_spender] != 0)) return false;`

However, this is just an attempt to modify user behaviour. If the user does attempt to change from one non-zero value to another, then the double spend can still happen, since the attacker will set the value to zero.

If desired, a nonstandard function can be added to minimize hassle for users. The issue can be fixed with minimal inconvenience by taking a change value rather than a replacement value:

```
function increaseAllowance (address _spender, uint256 _addedValue)
returns (bool success) {
  uint oldValue = approved[msg.sender][_spender];
  approved[msg.sender][_spender] = safeAdd(oldValue, _addedValue);
  return true;
}
```

Even if this function is added, it's important to keep the original for compatibility with the ERC20 specification.

Likely impact of this bug is low for most situations. This contract **DOES NOT** implement an `increaseAllowance` and a `decreaseAllowance` function*, thus it does not apply mitigation for approval double-spend attack*.

For more, see this discussion on GitHub:
https://github.com/ethereum/EIPs/issues/20#issuecomment263524729

- **Accidental Token Loss**
    - Token Smart Contracts should prevent transferring tokens to the token smart contract address if there's no good reason to not prevent, or if there's no way to take out tokens held by the token smart contract. The Yf-DAI smart contract *does not* prevent transferring of Yf-DAI to Yf-DAI smart contract address. If someone accidentally sends Yf-DAI to the Yf-DAI smart contract, their Yf-DAI will be locked up there forever with no way to take them out.

      This can be prevented by not allowing transfer to token smart contract address.

    - One more issue is when other ERC20 Tokens are transferred to the Yf-DAI smart contract, there would be no way to take them out, and this can be solved by implementing the "Any Token Transfer" improvement suggestion.

Issues Found

High Severity Issues

No high severity issues were found in the smart contract.

Moderate Severity Issues

No moderate severity issues were found in the smart contract.

Low Severity Issues

1. Yf-DAI *will be lost* / locked up with no way to take out, if sent to Yf-DAI smart contract.
   FIX: `require(to != address(this));` in `transfer` and `transferFrom` functions.

2. Any other ERC20 Tokens if sent to Yf-DAI smart contract address, will be locked up forever with no way to take out
   FIX: See the "*Any Token Transfer*" in improvements suggestions.

Both of the above issues are subjective, they can only happen if the user sends the tokens to the token smart contract addresses, though many times the user isn't aware or mistakenly copies wrong contract address, once sent, the tokens cannot be recovered if there's no way to take them out.

3. The smart contract does not apply mitigation for approval double-spend attack.

   FIX: Add `increaseAllowance` and `decreaseAllowance` for approval double-spend attack mitigation.

4. The smart contract does not prevent token transfers to self, which may lock up Yf-DAI in YfDAIfinance smart contract if someone accidentally sends them to the smart contract address. And the smart contract does not include a way for taking out ERC20 tokens from it which will result in any tokens sent this smart contract potentially irrevocably locked.

Line by line comments

- Line 5:
  The compiler version is specified as ^0.6.7, this means the code can be compiled with solidity compilers with versions greater than or equal to 0.6.7, that's totally fine, though it is recommended to remove the caret symbol while saving the code so that there are no unexpected issues with compiling in future when the compiler versions will be different.

- Lines 10 to 26:
  The `Owned` contract is included, which allows basic access management in smart contract. Ownership can be transferred in a two step manner, the owner executes `changeOwner` function with `_newOwner` address, and after that `_newOwner` accepts ownership of smart contract to complete the transfer, this is a very elegant way of ownership transfer.

- Lines 28 to 37:
  The ERC20 Solidity Interface is included as an abstract contract.

- Lines 39 to 74:
  `Token` contract is implemented inheriting from ERC20 and Owned contracts, containing all necessary ERC20 standard features. It checks for overflows in the `transfer` and `transferFrom` function which is pretty nice.

- Lines 76 to 91:
  `YfDAIfinance` contract is implemented inheriting from Token contract, it includes an external payable function `receive` to receive ether and forward them to contract owner. The constructor assigns the name as "YfDAI.finance", symbol as "Yf-DAI", and 18 decimals. It sets the contract owner as the contract deployer, assigns a total supply of 21,000 Yf-DAI and sets it as the owner's balance.

Improvement Suggestions

We suggest adding two more optional features.

- ## Ability to Reclaim Tokens transferred to Yf-DAI Smart contract

  If someone inadvertently sends ERC20 Tokens to a wrong contract address, and the contract does not have any mechanism to call `transfer` on an arbitrary token contract, the smart contract will never be able to claim any tokens held by itself and those tokens will be locked up in the smart contract forever.

  This can be avoided by adding a non-standard function to call `transfer` on an arbitrary token contract. That way, if someone inadvertently sends any ERC20 Tokens to this Token Contract, the owner will be able to call this function to transfer those tokens out of the contract.

  This can be done by adding the following (or similar) function to the `YfDAIfinance` contract.

```
function transferAnyERC20(address _tokenAddress, address _to, uint _amount)
    public onlyOwner {
      ERC20(_tokenAddress).transfer(_to, _amount);
  }
```

- ## One step transfer to contract

  Normally in ERC20 standard, when integrating the token with smart contracts or DAPPs for accepting payments, two steps are needed:

  1. The user approves the smart contract to transfer some amount their tokens on the user's behalf, using `approve` function.

  2. The smart contract then transfers the required tokens from the user's account to the beneficiary account and processes the payment.

  This way, normally two transactions are required and it spends more gas and more time than one transaction. It is recommended to include a one-step transfer to contract function to do the above process in one transaction and save time and gas fee.

  One step send (`approveAndCall`) can be implemented as follows:

```
interface tokenRecipient {
    function receiveApproval(address _from, uint256 _value, bytes calldata
_extraData) external;
}

contract YfDAIfinance is ERC20, Ownable {

    // … existing code here...


    // the one step transfer to contract function
    function approveAndCall(address _spender, uint256 _value, bytes
calldata_extraData)
        external
        returns (bool success)
    {
        tokenRecipient spender = tokenRecipient(_spender);
        if (approve(_spender, _value)) {
            spender.receiveApproval(msg.sender, _value, _extraData);
            return true;
        }
    }
}
```

Appendix

Smart Contract Summary

- Contract Owned
    - From Owned
        - acceptOwnership() (public)
        - changeOwner(address) (public)
- Contract ERC20
    - From ERC20
        - allowance(address,address) (public)
        - approve(address,uint256) (public)
        - balanceOf(address) (public)
        - transfer(address,uint256) (public)
        - transferFrom(address,address,uint256) (public)
- Contract Token
    - From Owned
        - acceptOwnership() (public)
        - changeOwner(address) (public)
    - From Token
        - allowance(address,address) (public)
        - approve(address,uint256) (public)
        - balanceOf(address) (public)
        - transfer(address,uint256) (public)
        - transferFrom(address,address,uint256) (public)
- Contract YfDAIfinance (Most derived contract)
    - From Token
        - allowance(address,address) (public)
        - approve(address,uint256) (public)
        - balanceOf(address) (public)
        - transfer(address,uint256) (public)
        - transferFrom(address,address,uint256) (public)
    - From Owned

        - acceptOwnership() (public)
        - changeOwner(address) (public)
    - From YfDAIfinance
        - constructor() (public)
        - receive() (external)

Slither Results

```
> slither YfDAIfinance.sol

Compilation warnings/errors on yfdai.sol:
Warning: SPDX license identifier not provided in source file. Before publishing,
consider adding a comment containing "SPDX-License-Identifier: <SPDX-License>" to
each source file. Use "SPDX-License-Identifier: UNLICENSED" for non-open-source
code. Please see https://spdx.org for more information.
--> yfdai.sol


INFO:Detectors:
Pragma version^0.6.7 (yfdai.sol#5) allows old versions
solc-0.6.12 is not recommended for deployment
Reference: https://github.com/crytic/slither/wiki/Detector-
Documentation#incorrect-versions-of-solidity
INFO:Detectors:
Parameter Owned.changeOwner(address)._newOwner (yfdai.sol#17) is not in mixedCase
Parameter Token.balanceOf(address)._owner (yfdai.sol#46) is not in mixedCase
Parameter Token.transfer(address,uint256)._to (yfdai.sol#48) is not in mixedCase
Parameter Token.transfer(address,uint256)._amount (yfdai.sol#48) is not in
mixedCase
Parameter Token.transferFrom(address,address,uint256)._from (yfdai.sol#56) is not
in mixedCase
Parameter Token.transferFrom(address,address,uint256)._to (yfdai.sol#56) is not in
mixedCase
Parameter Token.transferFrom(address,address,uint256)._amount (yfdai.sol#56) is
not in mixedCase
Parameter Token.approve(address,uint256)._spender (yfdai.sol#65) is not in
mixedCase
Parameter Token.approve(address,uint256)._amount (yfdai.sol#65) is not in
mixedCase
Parameter Token.allowance(address,address)._owner (yfdai.sol#71) is not in
mixedCase
Parameter Token.allowance(address,address)._spender (yfdai.sol#71) is not in
mixedCase
Reference: https://github.com/crytic/slither/wiki/Detector-
Documentation#conformity-to-solidity-naming-conventions
INFO:Detectors:
YfDAIfinance.constructor() (yfdai.sol#78-85) uses literals with too many digits:
        - totalSupply = 210000000000000000000000 (yfdai.sol#82)
Reference: https://github.com/crytic/slither/wiki/Detector-Documentation#too-many-
digits
INFO:Detectors:
changeOwner(address) should be declared external:
```

```
        - Owned.changeOwner(address) (yfdai.sol#17-20)
acceptOwnership() should be declared external:
        - Owned.acceptOwnership() (yfdai.sol#21-25)
balanceOf(address) should be declared external:
        - ERC20.balanceOf(address) (yfdai.sol#30)
        - Token.balanceOf(address) (yfdai.sol#46)
transfer(address,uint256) should be declared external:
        - ERC20.transfer(address,uint256) (yfdai.sol#31)
        - Token.transfer(address,uint256) (yfdai.sol#48-54)
transferFrom(address,address,uint256) should be declared external:
        - ERC20.transferFrom(address,address,uint256) (yfdai.sol#32)
        - Token.transferFrom(address,address,uint256) (yfdai.sol#56-63)
approve(address,uint256) should be declared external:
        - Token.approve(address,uint256) (yfdai.sol#65-69)
        - ERC20.approve(address,uint256) (yfdai.sol#33)
allowance(address,address) should be declared external:
        - ERC20.allowance(address,address) (yfdai.sol#34)
        - Token.allowance(address,address) (yfdai.sol#71-73)
Reference: https://github.com/crytic/slither/wiki/Detector-Documentation#public-
function-that-could-be-declared-external
INFO:Slither:yfdai.sol analyzed (4 contracts with 46 detectors), 21 result(s)
found
```

Purpose of the report

The Audits and the analysis described therein are created solely for Clients and published with their consent. The scope of our review is limited to a review of Solidity code and only the Solidity code we note as being within the scope of our review within this report. The Solidity language itself remains under development and is subject to unknown risks and flaws. The review does not extend to the compiler layer, or any other areas beyond the Solidity programming language that could present security risks. Cryptographic tokens and smart contracts are emergent technologies and carry with them high levels of technical risk and uncertainty.

The Audits are not an endorsement or indictment of any particular project or team, and the Audits do not guarantee the security of any particular project. This Report does not consider, and should not be interpreted as considering or having any bearing on, the potential economics of a token, token sale or any other product, service or other asset. Cryptographic tokens are emergent technologies and carry with them high levels of technical risk and uncertainty. No Report provides any warranty or representation to any Third-Party in any respect, including regarding the bugfree nature of code, the business model or proprietors of any such business model, and the legal compliance of any such business. No third party should rely on the Audits in any way, including for the purpose of making any decisions to buy or

sell any token, product, service or other asset. This Report does not constitute investment advice, is not intended to be relied upon as investment advice, is not an endorsement of this project or team, and it is not a guarantee as to the absolute security of the project. There is no owed duty to any Third-Party by virtue of publishing these Audits.

Disclaimer

The audit makes no statements or warranties about utility of the code, safety of the code, suitability of the business model, regulatory regime for the business model, or any other statements about fitness of the contracts to purpose, or their bug free status. The audit documentation is for discussion purposes only.

THE CONTENT OF THIS AUDIT REPORT IS PROVIDED "AS IS", WITHOUT REPRESENTATIONS AND WARRANTIES OF ANY KIND, AND BLOCKCHAIN CONSILIUM DISCLAIMS ANY LIABILITY FOR DAMAGE ARISING OUT OF, OR IN CONNECTION WITH, THIS AUDIT REPORT. COPYRIGHT OF THIS REPORT REMAINS WITH BLOCKCHAIN CONSILIUM.