defactor

SMART CONTRACT REVIEW

zokyo

November 15th 2023 | v. 1.0

Security Audit Score

PASS

Zokyo Security has concluded that this smart contract passes security qualifications to be listed on digital asset exchanges.

ZOKYO AUDIT SCORING DEFACTOR

1. Severity of Issues:
    - Critical: Direct, immediate risks to funds or the integrity of the contract. Typically, these would have a very high weight.
    - High: Important issues that can compromise the contract in certain scenarios.
    - Medium: Issues that might not pose immediate threats but represent significant deviations from best practices.
    - Low: Smaller issues that might not pose security risks but are still noteworthy.
    - Informational: Generally, observations or suggestions that don't point to vulnerabilities but can be improvements or best practices.
2. Test Coverage: The percentage of the codebase that's covered by tests. High test coverage often suggests thorough testing practices and can increase the score.
3. Code Quality: This is more subjective, but contracts that follow best practices, are well-commented, and show good organization might receive higher scores.
4. Documentation: Comprehensive and clear documentation might improve the score, as it shows thoroughness.
5. Consistency: Consistency in coding patterns, naming, etc., can also factor into the score.
6. Response to Identified Issues: Some audits might consider how quickly and effectively the team responds to identified issues.

HYPOTHETICAL SCORING CALCULATION:

Let's assume each issue has a weight:
- Critical: -30 points
- High: -20 points
- Medium: -10 points
- Low: -5 points
- Informational: -1 point

Starting with a perfect score of 100:
- Critical issues: 1 issue (resolved): 0 points deducted
- High issues: 1 issue (verified): -5 points deducted. While the issue is verified, it still raises concerns from auditor's side regarding the absence of the reliable oracle
- 0 Medium issues: 0 points deducted
- Low issues: 1 issue (verified): 0 points deducted
- Informational issues: 7 issues (3 resolved, 4 verified): -3 points deducted due to that (as still even verified issues create a concerns regarding the efficiency), to the failed check against the backdoor and including the lack of documentation.

Thus, 100 - 5 - 3 = 92

TECHNICAL SUMMARY

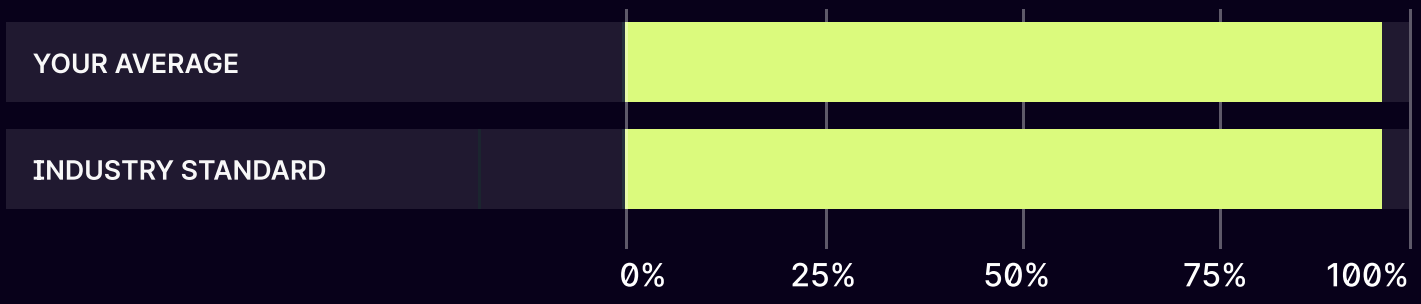
This document outlines the overall security of the Defactor smart contract evaluated by the Zokyo Security team.

The scope of this audit was to analyze and document the Defactor smart contract codebase for quality, security, and correctness.

Contract Status

There was 1 critical issue found during the audit. (See Complete Analysis)

Testable Code



98.8% of the code is testable, which is above the industry standard of 95%.

It should be noted that this audit is not an endorsement of the reliability or effectiveness of the contracts but rather limited to an assessment of the logic and implementation. In order to ensure a secure contract that can withstand the Ethereum network's fast-paced and rapidly changing environment, we recommend that the Defactor team put in place a bug bounty program to encourage further active analysis of the smart contract.

Table of Contents

AUDITING STRATEGY AND TECHNIQUES APPLIED

The source code of the smart contract was taken from the Defactor repository:
Repo: https://github.com/defactor-com/buyback

Last commit: cb45517c3bea4f3ba4c31cf1e5dad55547fcc0b2

Within the scope of this audit, the team of auditors reviewed the following contract:

- Buyback.sol

During the audit, Zokyo Security ensured that the contract:

- Implements and adheres to the existing standards appropriately and effectively;
- The documentation and code comments match the logic and behavior;
- Distributes tokens in a manner that matches calculations;
- Follows best practices, efficiently using resources without unnecessary waste;
- Uses methods safe from reentrance attacks;
- Is not affected by the most resent vulnerabilities;
- Meets best practices in code readability, etc.

Zokyo Security has followed best practices and industry-standard techniques to verify the implementation of Defactor smart contract. To do so, the code was reviewed line by line by our smart contract developers, who documented even minor issues as they were discovered. Part of this work includes writing a test suite using the Hardhat testing framework. In summary, our strategies consist largely of manual collaboration between multiple team members at each stage of the review:

01	Due diligence in assessing the overall code quality of the codebase.	03	Testing contracts logic against common and uncommon attack vectors.
02	Cross-comparison with other, similar smart contracts by industry leaders.	04	Thorough manual review of the codebase line by line.

Executive Summary

Zokyo Security received a contract from the Defactor team, which included a buyback contract.

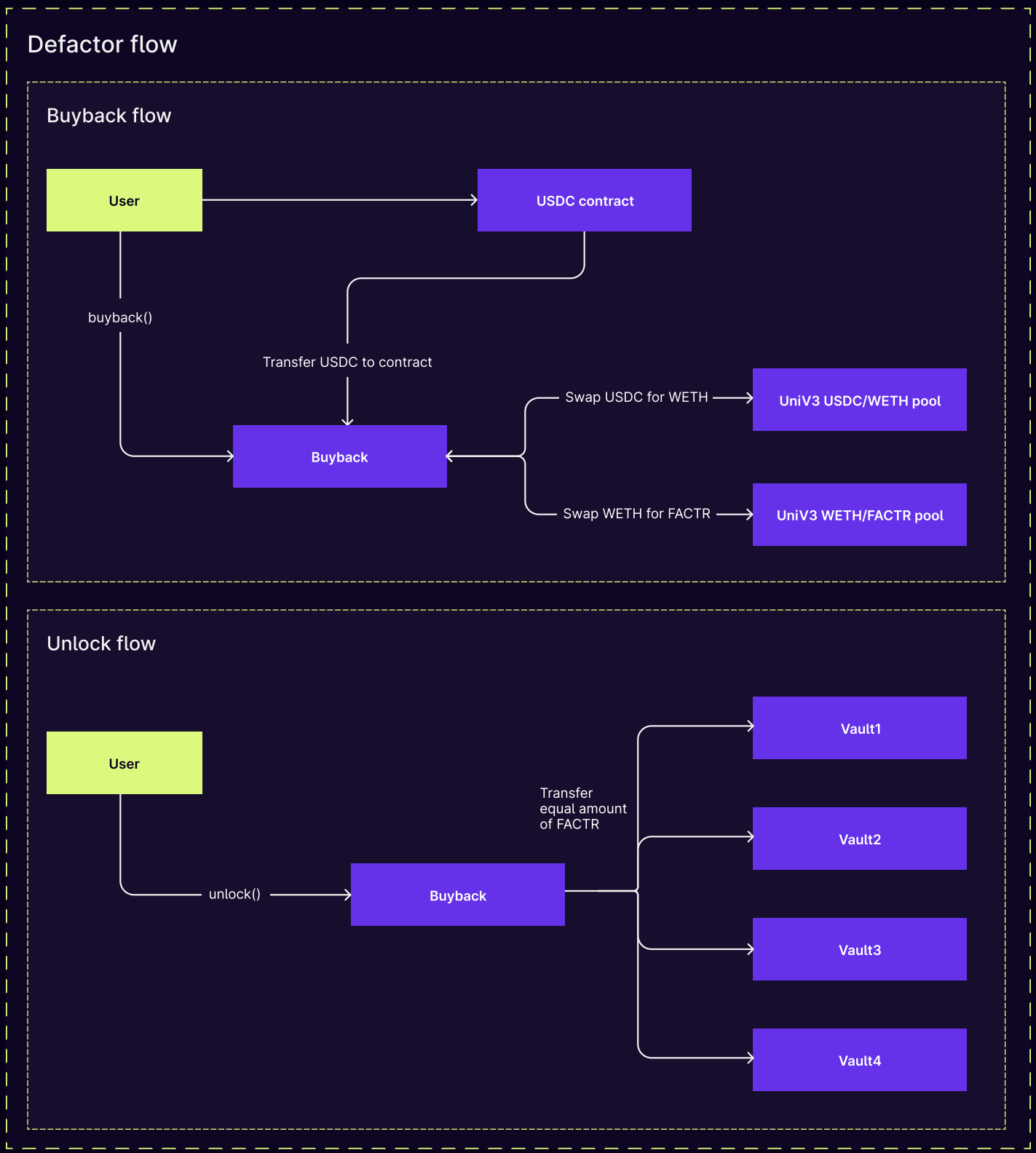
The audit's goal was to verify the correctness of buyback and unlock functions and check that the tokens on the contract were safe. The auditors checked the code line by line, compared it against their own checklist of vulnerabilities, validated the business logic of the contracts, and ensured that best practices in terms of gas spending were applied. The setup and deployment scripts of the contracts were also carefully audited.

During the manual portion of the audit, a critical issue was discovered regarding the incorrect calculation of tokens to be unlocked. Additionally, a significant issue was identified concerning a potential flash loan attack. The Defactor team resolved the critical issue and confirmed with our security team that due to UniSwapV3, the transaction would revert if a substantial change were made to the token price. Other issues of low-level and informational severity were related to a lack of documentation, code structure, pragma version, magic numbers, withdrawal array, and hardcoded pool fees. The Defactor team resolved most of these issues, while others were discussed and verified.

Another part of the audit process involved examining the native tests prepared by the Defactor team and preparing a set of custom test scenarios, such as a flash loan attack. It should be noted that the Defactor team provided basic tests to cover contract logic. However, Zokyo Security prepared its own set of unit tests and additional scenarios to cover the complex functionality of Defactor contracts. The complete set of unit tests can be found in the Code Coverage and Test Result sections.

Overall, the security level of the contracts is high. The contract is well-written, thoroughly tested, and carefully audited. Nevertheless, the contract fails the check against the backdoors, as the contracts is upgradeable. Despite it is a common approach, it still creates a controllable backdoor, which should be noted for the funds bearing contract.

Protocol overview



DEFACTOR (BUYBACK) DESCRIPTION

Roles and Responsibilities

No roles present in contracts

Settings

Every contract that is needed for correct contract use are stored in buyback.sol contract variables.
List of contracts:

- vault(1-4): multisig wallets, where FACTR tokens will be transferred.
- uniswapRouter: UniSwapV3 router address.
- FACTR: Defactor token address.
- USDC: Circle USD token address.
- WETH: Wrapped ETH token address, that gets from UniSwapV3 router.
- factory: UniSwapV3 Factory address, that gets from UniSwapV3 router.

It is necessary to check addresses before deployment, especially for vault1, vault2, vault3, and vault4, as these are the addresses where FACTR tokens will be sent.

Deployment script

Deployment script is located at **scripts/deploy.js**

All the addresses needed are already located in the contract code. Deployment script uses hardhat upgrades to deploy proxy. Based on contract addresses that are in the buyback.sol contract, the network for deployment is Ethereum mainnet. **Addresses that are coded in the buyback contract should be validated before deployment.**

List of values assets

- USDC token - used to swap for FACTR token.
- FACTR token - stored on contract until it unlocked and transferred to vaults.

STRUCTURE AND ORGANIZATION OF THE DOCUMENT

For the ease of navigation, the following sections are arranged from the most to the least critical ones. Issues are tagged as “Resolved” or “Unresolved” depending on whether they have been fixed or addressed. The issues that are tagged as “Verified” contain unclear or suspicious functionality that either needs explanation from the Client or remains disregarded by the Client. Furthermore, the severity of each issue is written as assessed by the risk of exploitation or other unexpected or otherwise unsafe behavior:

Critical

The issue affects the contract in such a way that funds may be lost, allocated incorrectly, or otherwise result in a significant loss.

High

The issue affects the ability of the contract to compile or operate in a significant way.

Medium

The issue affects the ability of the contract to operate in a way that doesn’t significantly hinder its behavior.

Low

The issue has minimal impact on the contract’s ability to operate.

Informational

The issue has no impact on the contract’s ability to operate.

COMPLETE ANALYSIS

CRITICAL-1 | RESOLVED

Wrong balance check of FACTR token.

buyback.sol: buyback(), line 102.
After a swap, the FACTR token balance is checked for the contract, not for the swapped tokens. In this case, an incorrect number of tokens is being written to the withdrawals array. For instance, if the first buyback is invoked and 100 tokens are swapped, these are added to the first element of the withdrawals array. If a second buyback is invoked and another 100 tokens are swapped, 200 tokens are added to the second element of the withdrawal array because it checks the balance of the contract. To resolve this, add a check for the balance before and after the swap and subtract to get the exact swapped amount to add to the withdrawal array.

Recommendation:

Check balance before and after swap to have exact swapped amount.

Post-audit:

Added balance check before and after the swap.

HIGH-1 | VERIFIED

Potential for Flash Loan Attack during buyback execution.

Buyback.sol: buyback() function.
In the Buyback contract, the buyback() function uses the price of the FACTR token obtained from the Uniswap V3 Pool to check the condition that the price has not fallen by more than 1%. However, this price can be artificially maintained at an inflated level for 100 seconds (the time used to obtain the historical price). This could be done using bots capable of supporting the price, enabling a Flash Loan attack.

Recommendation:

Use either on-chain TWAP oracles to validate the price retrieved from AMM **OR** off-chain oracles such as Chainlink **OR** provide the minimum output amount as a function parameter.

Post-audit:

After team discussion, 100 seconds for the Uniswap V3 pool should be enough to avoid a price increase/decrease. Though still, oracle absence raises a concern from auditor's side.

LOW-1 | VERIFIED

Use the latest pragma version.

Using the latest version of pragma will make the contract more secure as the new versions fix known security issues.

Recommendation:
Change contracts to 0.8.21 version.

Post-audit:
Added balance check before and after swap.

INFORMATIONAL-1 | RESOLVED

NatSpec required.

NatSpec is a documentation format that helps in the readability of contract functions and a better understanding of what they are doing. It is recommended to have NatSpec in the contract. More details about it here: https://docs.soliditylang.org/en/v0.8.21/NatSpec-format.html.

Recommendation:
Add NatSpec and comments to contract functions and variables. Also, add NatSpec to the contract description to emphasize the contract role.

Post-audit:
Basic NatSpec for functions added to the contract.

INFORMATIONAL-2 | RESOLVED

Contract structure.

Variable `withdrawals` goes after the initialize function. It is recommended to have all variables before any function in the contract to improve its readability. More information can be found here: https://docs.soliditylang.org/en/latest/style-guide.html#order-of-layout

Recommendation:
Move `withdrawals` variable and `Withdrawal` struct before initialize function.

Post-audit:
`withdrawals` was moved before `Withdrawal` struct.

INFORMATIONAL-3 | RESOLVED

Magic numbers in the contract.

It is recommended to have numbers in the contract as constant variables. This increases the readability of the code.

Recommendation:
Make numbers such as `300`, `500`, `1000`, etc., as constant variables rather than using the values directly.

Post-audit:
Numbers moved to contract variables.

INFORMATIONAL-4 | VERIFIED

Elements order in withdrawals array.

The current logic could change the order in the withdrawals array. For example, an array has four elements [1, 2, 3, 4]. If invoking the unlock() function to remove the first element (unlock(0)), the array will be [4, 2, 3]. Verify if the order matters for the withdrawals array.

Recommendation:
Verify if the order of elements in the `withdrawals` array is irrelevant **OR** implement a mechanism where the order won't be changed.

Post-audit:
The Defactor team verified that it is the most efficient way to interact with the array.

INFORMATIONAL-5 | VERIFIED

Add check for array length.

buyback.sol: unlock(), line 118.
As for now, to remove an element from the withdrawals array, it rewrites the removed element with the last and pops the last element. If only one element is in an array, it is an unnecessary step, as the result should just be an empty array. Consider adding a check for array length to check if one element is left in the array.

Recommendation:
Add a length check to check if one element is left in the array.

Post-audit:
The Defactor team affirmed that such a solution would only increase the complexity of the contract logic. Though still the efficiency of current mechanism can be improved.

INFORMATIONAL-6 | VERIFIED

Lack of function for token rescuing.

The contract does not have a function to withdraw accidentally sent tokens.
If tokens are accidentally sent to the contract, they will be stuck forever. However, care should be taken to ensure that the contract's USDC and FACTR tokens cannot be withdrawn

Recommendation:
Add a function that allows the owner of the contract to withdraw accidentally sent tokens. Ensure that this function cannot be used to withdraw the contract's USDC and FACTR tokens.

Post-audit:
The Defactor team verified that because the contract is upgradable, they could add functionality to extract tokens sent by mistake.

INFORMATIONAL-7 | VERIFIED

Hardcoded fee value for the pool.

The contract uses a hardcoded value for the pool fee when interacting with Uniswap pools in the `buyback()` function. If the pool fee changes or the contract needs to interact with a pool with a different fee, the contract will not work as expected.

Recommendation:
Consider making the pool fee a variable that the contract owner can update. This would add flexibility to interact with different Uniswap pools. However, it is essential to clarify whether the contract is intended to work only with Uniswap pools with specific fees. If the contract needs to interact with pools with different fees, an option to reset the pool might be necessary.

Post-audit:
The Defactor team verified that in UniSwapV3, the fee will not be changed.

	buyback.sol
Re-entrancy	Pass
Access Management Hierarchy	Pass
Arithmetic Over/Under Flows	Pass
Unexpected Ether	Pass
Delegatecall	Pass
Default Public Visibility	Pass
Hidden Malicious Code	Pass
Entropy Illusion (Lack of Randomness)	Pass
External Contract Referencing	Pass
Short Address/ Parameter Attack	Pass
Unchecked CALL Return Values	Pass
Race Conditions / Front Running	Pass
General Denial Of Service (DOS)	Pass
Uninitialized Storage Pointers	Pass
Floating Points and Precision	Pass
Tx.Origin Authentication	Pass
Signatures Replay	Pass
Pool Asset Security (backdoors in the underlying ERC-20)	Fail

CODE COVERAGE AND TEST RESULTS FOR ALL FILES

Tests written by Zokyo Security

As a part of our work assisting Defactor in verifying the correctness of their contract code, our team was responsible for writing integration tests using the Hardhat testing framework.

The tests were based on the code's functionality and a review of the Defactor contract requirements for details about issuance amounts and how the system handles these.

```
Buyback
    Initialization
    ✓ Should initialize correctly (268ms)
    Estimation
    ✓ Should estimate DAI amount out (639ms)
    ✓ Should estimate USDC amount out (137ms)
    Buyback
    ✓ Should perform buyback (6004ms)
    ✓ Shouldn't buyback if balance insufficient (303ms)
    ✓ Should increase FACTR price
    Unlock
    ✓ Should unlock (1413ms)
    ✓ Unlock should revert if 1 year has not passed (682ms)

    8 passing (14s)
```

The resulting code coverage (i.e., the ratio of tests-to-code) is as follows:

FILE	% STMTS	% BRANCH	% FUNCS
Buyback.sol	100	72.22	100

CODE COVERAGE AND TEST RESULTS FOR ALL FILES

Tests written by Defactor

As a part of our work assisting Defactor in verifying the correctness of their contract code, our team has checked the complete set of tests prepared by the Defactor team.

We must mention that the original code has significant original coverage with testing scenarios provided by the Defactor team. All of them were also carefully checked by the team of auditors.

```
buyback
    Deployment
    ✓ estimateAmountOut should return the correct value (213ms)
    ✓ Should estimate DAI amount out correctly
    ✓ Should estimate USDC amount out correctly
    ✓ Should perform buyback correctly (144ms)
    ✓ Unlock should revert if 1 year has not passed (108ms)
    ✓ Unlock should succeed (126ms)
    ✓ Transaction should revert if token price fell for over 1%. (135ms)

    8 passing (14s)
```

The resulting code coverage (i.e., the ratio of tests-to-code) is as follows:

FILE	% STMTS	% BRANCH	% FUNCS
Buyback.sol	93.1	50	100

We are grateful for the opportunity to work with the Defactor team.

The statements made in this document should not be interpreted as an investment or legal advice, nor should its authors be held accountable for the decisions made based on them.

Zokyo Security recommends the Defactor team implement a bug bounty program to encourage further analysis of the smart contract by third parties.

zokyo