SMART CONTRACTS REVIEW

zokyo

March 28th 2025 | 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.

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:
- 0 Critical issues: 0 points deducted
- 0 High issues: 0 points deducted
- 0 Medium issues: 0 points deducted
- 5 Low issues: 5 acknowledged = - 5 points deducted
- 3 Informational issues: 3 acknowledged = 0 points deducted

Thus, 100 - 5 = 95

TECHNICAL SUMMARY

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

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

Contract Status

There were 0 critical issue found during the audit. (See Complete Analysis)

It should be noted that this audit is not an endorsement of the reliability or effectiveness of the contract/s 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/s.

Table of Contents

AUDITING STRATEGY AND TECHNIQUES APPLIED

The Smart contract's source code was taken from the Defactor repository.
Repo: https://github.com/defactor-com/blockchain

Last commit - 3af91f92c34d67e3a8bc4129bb18e52965cd0a04

Contracts under the scope:

- ./buyback/buyback.storage.sol
- ./buyback/buyback.sol
- ./buyback/buyback.interface.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 recent 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/s. To do so, the code was reviewed line by line by our smart contract developers, who documented even minor issues as they were discovered. 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.

02 Cross-comparison with other, similar smart contract/s by industry leaders.

03 Testing contract/s logic against common and uncommon attack vectors.

Executive Summary

The Defactor protocol aims to innovate at the intersection of real world assets and decentralized finance. Defactor's FACTR token is a utility token that drives the technology's ecosystem. The revenue generated goes into the buyback contract supporting adoption. More usage of the ecosystem means more buybacks allowing growth to be aligned with token demand. In Defactor, there exists their core pillars including technology & innovation by pioneering innovation in RWAs and DeFI, governance allowing a collective group of individuals to collaborate and drive innovation in RWA tokenization, community to create a global support system and liquidity & markets which ensures the constant availability of market liquidity in addition to monitoring market conditions which is what this set of smart contracts addresses.

Zokyo was tasked with conducting the security reviews for the buyback contracts. The users can interact with the Defactor protocol in a multitude of ways whether it be for staking to stake various tokens and earn rewards, using the pools to allow users to create, manage and interact with funding pools using the most popular token standards such as ERC20, ERC721 or ERC1155, in addition to facilitating lending and borrowing positions through the ERC20 collateral pools.

Overall, the code is well thought out and well engineered in accordance with Defactor's guidelines and addresses a growing area in our industry however, additional natspec within the code itself could add to the code presentation and added context. The security team discovered issues ranging from Critical issues all the way down to information that aims to address best engineering practices. These issues revolved around root causes which stemmed from errors in calculations in addition to some logic errors, the way fees are handled and validation around return values from Chainlink oracles and reinforcement of best engineering practices such as Checks Effects and Interactions (CEI), and ensuring functions return key variables. Some issues were deemed to be within the protocol's risk appetite resulting in acknowledgement but still remained in this report to keep the users informed of the security review process.

After extensive discussions around the discovered issues, the protocol team promptly fixed the issues where a fix review took place to ensure that no additional bugs were introduced into the code base as a result of the fixes and to ensure that the discovered attack vectors were no longer exploitable. Whilst some unit testing does exist using the hardhat framework, there can always be more testing introduced to ensure the intended actions of the contracts in addition to fuzz testing to catch any edgecase scenarios. In addition to this, it can also be helpful to update the documentation on the Defactor development documents so that users can review the contracts intended purpose. As mentioned previously, natspec within the contracts themselves can also assist with this. Finally, a bug bounty program is also highly advised for the protocol's continuous commitment to ensuring security for the ecosystem and its users. We at Zokyo wish Defactor all the best of luck in their deployment to the relevant production blockchains!

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” or “Acknowledged” depending on whether they have been fixed or addressed. Acknowledged means that the issue was sent to the Defactor team and the Defactor team is aware of it, but they have chosen to not solve it. 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

FINDINGS SUMMARY

#	Title	Risk	Status
1	Hardcoded addresses within `buyback.storage.sol` can be used to execute malicious attacks	High	Resolved
2	If there is a deviation greater than 1% downward, funds will get stuck forever	Low	Acknowledged
3	`USDC` will get locked within `buyback.sol` if threshold is not met	Low	Acknowledged
4	Uniswap's quoter calls can revert due to gas consumption, leading to a denial of service	Low	Acknowledged
5	Hardcoded `minAmountOut` to zero can lead to experiencing slippage	Low	Acknowledged
6	Improper Slippage Protection Due to Block Displacement Attack	Low	Acknowledged
7	It is not checked whether certain tokens are pool tokens or not	Informational	Acknowledged
8	CEI pattern not followed	Informational	Resolved
9	Hardcoded Swap Path Causing Potential Inefficiency	Informational	Acknowledged

Hardcoded addresses within `buyback.storage.sol` can be used to execute malicious attacks

Description:
The `buyback.storage.sol` smart contract contains several hardcoded addresses:

address public constant uniswapRouter = 0xE592427A0AEce92De3Edee1F18E0157C05861564;
address public constant FACTR = 0xe0bCEEF36F3a6eFDd5EEBFACD591423f8549B9D5;
address public constant USDC = 0xA0b86991c6218b36c1d19D4a2e9Eb0cE3606eB48;
address public constant WETH = 0xC02aaA39b223FE8D0A0e5C4F27eAD9083C756Cc2;
IQuoterV2 constant QUOTER = IQuoterV2(0×61fFE014bA17989E743c5F6cB21bF9697530B21e);

These are the corresponding addresses for the Ethereum network. However, the team confirmed that the protocol is going to be deployed to 'ETH, BASE, Polygon, BSC, Layer2 on OP stack (like OP)', where these addresses are not correct.

Impact:
If the protocol is deployed with these hardcoded addresses to any network different than Ethereum it can result that the used address corresponds to a malicious smart contract which executes some unexpected behavior affecting the protocol in different ways.

Recommendation:
Do not use any hardcoded address, instead, set them within the` __buybackStorage_init()` function.

If there is a deviation greater than 1% downward, funds will get stuck forever

Description:
The `_sandwichProtection()` function within the `buyback.sol` smart contract is used to check if the current pool price is manipulated in comparison to the retrieved price from UniswapV3's TWAP oracle. This function only allows price to be deviated 1% downwards from the average price for the last 100 seconds:

```
function _sandwichProtectionCheck(address pool1, address pool2) view
private {

        uint wethAveragePrice = estimateAmountOut(USDC, uint128(10 **
IERC20(USDC).decimals()), WETH, ONE_HUNDRED, pool1);
        uint factrAveragePrice = estimateAmountOut(WETH,
uint128(wethAveragePrice), FACTR, ONE_HUNDRED, pool2);

        (,int24 currentTickPool1,,,,,) = IUniswapV3Pool(pool1).slot0();
        (,int24 currentTickPool2,,,,,) = IUniswapV3Pool(pool2).slot0();

        uint wethCurrentPrice =
OracleLibrary.getQuoteAtTick(currentTickPool1, uint128(10 **
IERC20(USDC).decimals()), USDC, WETH);
        uint factrCurrentPrice =
OracleLibrary.getQuoteAtTick(currentTickPool2, uint128(wethCurrentPrice),
WETH, FACTR);

        if (factrCurrentPrice < factrAveragePrice) {
            uint diff = factrAveragePrice - factrCurrentPrice;
            require(diff * ONE_HUNDRED / factrAveragePrice <= 1, "Price
has fallen more than 1%");
        }
    }
```

Impact:
If the price remains deviated more than 1% downwards from the average price of the last 100 seconds, it will be not possible to extract funds from the contract. This can happen in several situations where the price of an asset is decreasing rapidly in turbulent market conditions.

From the Uniswap v3 Oracles blog (https://blog.uniswap.org/uniswap-v3-oracles), it is observed that to shift a 30-minute TWAP by 20%, an attacker would need to manipulate prices across at least 30 consecutive blocks. Achieving this level of manipulation would require a validator to control 40% of block production within a 150-block window.

With secondsAgo set to only 100 seconds, an attacker with sufficient control could manipulate the TWAP for over 100 seconds and shift the spot price by 10%, for example. If an attacker is able to manipulate the TWAP price for this period, he is also able to manipulate the spot price, this allows them to keep the price deviation within the 1% threshold (but this a manipulated price), leading to a manipulated pool price being used for swaps.

Recommendation:
Instead of hardcoding the deviation threshold to 1% and the `secondsAgo` parameter to `100`, include two `onlyOwner` functions that allow modification of these parameters when needed. This will provide flexibility to adjust the values dynamically based on market conditions or security requirements.

Developer comments:
The `30` min TWAP could introduce a DOS issue, since the price can deviate alot, over 1%, on a larger scale and if so we wouldnt be able to perform a swap if the time window is too great. And secondly the buyback amount is normally `1000` USDC (min limit). If an attacker would want to extract any value, he would have to make the price rise on all the pools (otherwise arbitrage MEV bots will simply extract his value), for which he would need a lot of capital and even if he achieves that, users can sell their tokens, via limit order or just market orders. So doing all the for a couple hundred usdc is not economically viable. So it seems like there is no real benefit for an attacker to perform this kind of an attack as he would in most cases loose money.

`USDC` will get locked within `buyback.sol` if threshold is not met

Description:
The `Buyback.sol` smart contract includes a requirement within the `buyback()` function that the contract balance must reach at least `1000` USDC before executing a buyback. If this threshold is not met, the funds remain locked within the contract, preventing further operations.

```
require(usdcBalanceBefore >= ONE_THOUSAND * 10 **
IERC20(USDC).decimals(), "USDC Balance should be at least 1000");
```

Impact
It will not be possible to execute buybacks if the contract balance remains below `1000` USDC. This means that funds will be locked within the `buyback.sol` smart contract until the threshold is reached. The severity of this issue is categorized as medium since it can be resolved by directly sending more USDC to the contract.

Recommendation:
Consider implementing a mechanism to allow any amount for executing the buyback or implement an `onlyOwner` function that modifies the selected threshold.

Developer comments:
This is by design. Minimal amount on contract is `1000` USDC for buyback to be possible. Untill it is reached the funds will wait on the contract. Otherwise user can top up the balance and perform the buyback.

Uniswap's quoter calls can revert due to gas consumption, leading to a denial of service

Description:
The `calculateOptimalAmount()` function within the `buyback.sol` smart contract interacts with uniswap quoter contract when `providedOptimalAmount` is set to `0`.

```
function calculateOptimalAmount(bytes memory path, address pool1, address
pool2, uint maxAmount) public returns(uint maxAcceptableAmount) {

        while ( true ) {
            (uint quoterAmount, , , ) = QUOTER.quoteExactInput( path,
maxAmount
            uint twapOptimalAmount =
getOptimalTwapAmountThreshold(maxAmount, pool1, pool2);
            if(quoterAmount >= twapOptimalAmount) {
                return maxAmount;
            } else {
                maxAmount -= maxAmount / 10;
            }
        }
    }
```

However, the Uniswap Quoter can be gas-intensive and is not optimized for on-chain use, as its instability may lead to unexpected reversions. Uniswap actually recommends using it in backend applications. Docs (https://docs.uniswap.org/contracts/v3/reference/periphery/lens/Quoter):

```
These functions are not gas efficient and should not be called on chain.
```

Impact:
If the block gas limit is reached, the transaction will revert, leading to a denial of service for the functions calling `calculateOptimalAmount()`.

Recommendation:
The retrieved `optimalAmount` is utilized to compute the `minAmountOut` required for the swap. It is advisable to perform this calculation off-chain and pass it as a function parameter. Additionally, an onlyOwner modifier should be applied to the function to prevent users from setting `minAmountOut` to zero, mitigating the risk of fund loss.

Developer comments:
This is the only way to make this work. It is gas intensive indeed, but from out analysis it wouldn't reach block size and in most cases it would find the optimalAmount quite quickly.

Hardcoded `minAmountOut` to zero can lead to experiencing slippage

Description:
The `customBuyback()` function within the `buyback.sol` smart contract executes a call to `_executeSwap()` in order to perform a swap. This function receives the `minAmountOut` parameter which is hardcoded to `0`, this means that the swap can return `0` tokens in exchange without reverting:

```
_executeSwap(optimalAmount, path, 0);
```

Impact:
As `minAmountOut` has been hardcoded to zero, the swap can return `0` tokens in exchange without reverting. The severity of this issue has been selected as medium due to the protection for the sandwich attack implemented, otherwise it would have been set to High. However, as the protocol is going to be deployed to different blockchains and the `secondsAgo` parameter for the twap call is recommended to be a customizable parameter, there is still a possibility for the twap to get manipulated, specially for low liquidity pools.

Recommendation:
It is recommended to set `minAmountOut` as a parameter for the `customBuyback()` function so it is calculated off-chain and can not get manipulated. Additionally, an onlyOwner modifier should be applied to the function to prevent users from setting deadline to block.timestamp again.

Developer comments:
We have this covered by checking the average price via the oracle and enforcing the slippage there. So if amountOut is 1% less than average, transaction will revert.

Improper Slippage Protection Due to Block Displacement Attack

Description:
The `_executeSwap()` function from the `buyback.sol` smart contract sets the `deadline` parameter as `block.timestamp + THREE_HUNDRED`. This approach makes the swap vulnerable to a block displacement (or transaction reordering) attack, where a malicious actor could manipulate the transaction timing to exploit price slippage

```
function _executeSwap(uint amountIn, bytes memory path, uint
quoterAmount) private {

        ISwapRouter.ExactInputParams memory params = ISwapRouter
            .ExactInputParams({
            path: path,
            recipient: address(this),
            deadline: block.timestamp + THREE_HUNDRED,
            amountIn : amountIn,
            amountOutMinimum: quoterAmount
        });

        ISwapRouter(uniswapRouter).exactInput(params);
    }
```

Impact:
Setting the deadline to `block.timestamp + THREE_HUNDRED`, is in practice, the same as setting it to `block.timestamp`. This may affect the transaction to get displaced in time so a worse price is selected for the swap execution.

Recommendation:
The deadline selected for the swap is recommended to be introduced as a function parameter, so it can not get manipulated. Additionally, an `onlyOwner` modifier should be applied to the function to prevent users from setting the `deadline` to block.timestamp again.

Developer comments:
To ensure the amountOut is fair, we use the oracles, therefore if the amountOut would be 1% less than average (which is our maxSlippage), the transaction will fail anyhow

It is not checked whether certain tokens are pool tokens or not.

Description:
The `getOptimalTwapAmountThreshold()` function within the `buyback.sol` smart contract, receives by parameter `pool1` and `pool2` assuming that these pools are `FACTR-WETH` and `WETH- USDC` pools but not checking it:

```
function getOptimalTwapAmountThreshold(uint amountIn, address pool1,
address pool2) view public returns(uint) {
                uint wethPriceInFactr = estimateAmountOut(FACTR,
uint128(1e18), WETH, ONE_HUNDRED, pool2);
        uint factrPriceInUsdc = estimateAmountOut(WETH,
uint128(wethPriceInFactr), USDC, ONE_HUNDRED, pool1);

        return (amountIn * (10 ** (IERC20(FACTR).decimals()) /
factrPriceInUsdc)) *
            (TEN_THOUSAND - MAX_LIQUIDITY_SLIPPAGE) / TEN_THOUSAND;
    }
```

Impact:
Using incorrect pools can lead to unexpected behaviors.

Recommendation:
Implement a check to ensure that these pools contains the expected tokens.

Developer comments:
It is up to deployer to check and set these when deploying.

CEI pattern not followed

Description:
The `buybackWithdraw()` and `customBuybackWithdraw()` functions perform several external transfers of `FACTR` tokens before updating the contract state:

```
for (uint i; i < customBuybacks[id].distributionArray.length; i++) {
            require(
                IERC20(FACTR).transfer(
                    customBuybacks[id].distributionArray[i].account,
                    customBuybacks[id].buyAmount *
customBuybacks[id].distributionArray[i].bps / TEN_THOUSAND
                ), "TRANSFER_FAIL"
            );
        }

customBuybacks[id].withdrawn = true;
```

Impact:
There should be no security implications as the `FACTR` token is an already known and trusted tokens, otherwise the severity of the issue would have been higher. Nevertheless, it is always recommended to follow the CEI pattern.

Recommendation:
Follow the CEI pattern:

`customBuybackWithdraw()`:

```
function customBuybackWithdraw(uint id) external {

        require(customBuybacks[id].timeLocked + 365 days <
block.timestamp, "Unlock period not finished");
        require(customBuybacks[id].buyAmount != 0, "This is not a valid
buyback");
        require(!customBuybacks[id].withdrawn, "Unlock already
withdrawn");
 customBuybacks[id].withdrawn = true;
```

```
        for (uint i; i < customBuybacks[id].distributionArray.length; i+
+) {
            require(
                IERC20(FACTR).transfer(
                    customBuybacks[id].distributionArray[i].account,
                    customBuybacks[id].buyAmount *
customBuybacks[id].distributionArray[i].bps / TEN_THOUSAND
                ), "TRANSFER_FAIL"
            );
        }
        emit CustomWithdraw(id);
    }
```

• `buybackWithdraw()`:

```
function buybackWithdraw(uint id) external {

        require(buybacks[id].timeLocked + 365 days < block.timestamp,
"Unlock period not finished");
        require(buybacks[id].buyAmount != 0, "This is not a valid
buyback");
        require(!buybacks[id].withdrawn, "Unlock already withdrawn");

        uint amountPart = buybacks[id].buyAmount / 4;

        require(IERC20(FACTR).transfer(vault1, amountPart),
"TRANSFER_FAIL");
        require(IERC20(FACTR).transfer(vault2, amountPart),
"TRANSFER_FAIL");
        require(IERC20(FACTR).transfer(vault3, amountPart),
"TRANSFER_FAIL");
        require(IERC20(FACTR).transfer(vault4, buybacks[id].buyAmount -
(3 * amountPart)), "TRANSFER_FAIL");

        buybacks[id].withdrawn = true;

        emit Withdraw(id);
    }
```

Hardcoded Swap Path Causing Potential Inefficiency

Description:
The swap path is hardcoded as `USDC → WETH → FACTR`, potentially leading to suboptimal swap execution if a more direct or efficient path (e.g., a direct `USDC → FACTR` pool) becomes available or gains better liquidity.

Proof of Concept
A new liquidity pool with better rates and lower slippage becomes available (such as a direct USDC-FACTR pair), but the protocol continues using the fixed, indirect swap path. This results in unnecessary slippage and increased costs.

Recommendation:
Implement dynamic path selection logic or allow governance-controlled path adjustments. Consider integrating routing solutions or external oracles to determine the optimal swap path automatically, ensuring the most efficient swap execution.

Developer comments:
Acknowledged

	./buyback/buyback.storage.sol ./buyback/buyback.sol ./buyback/buyback.interface.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

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