Zellic

Fractal Protocol

Smart Contract Security Assessment

March 18, 2022

Prepared for:

Alex Elkrief

Fractal Protocol

Prepared by:

Jasraj Bedi and Ayaz Mammadov

Zellic Inc.

Contents

About Zellic

Zellic was founded in 2020 by a team of blockchain specialists with more than a decade of combined industry experience. We are leading experts in smart contracts and Web3 development, cryptography, web security, and reverse engineering. Before Zellic, we founded perfect blue, the top competitive hacking team in the world. Since then, our team has won countless cybersecurity contests and blockchain security events.

Zellic aims to treat clients on a case-by-case basis and to consider their individual, unique concerns and business needs. Our goal is to see the long-term success of our partners rather than to simply provide a list of present security issues. Similarly, we strive to adapt to our partners' timelines and to be as available as possible. To keep up with our latest endeavors and research, check out our website zellic.io or follow @zellic_io on Twitter. If you are interested in partnering with Zellic, please email us at hello@zellic.io or contact us on Telegram at https://t.me/zellic_io.

1   Introduction

1.1   About Fractal Protocol

Fractal is a cross-chain liquidity routing protocol. Its first product is USDF, a yield bearing stable coin which accrues value at a fixed APR. The Fractal yield is powered by a diversified set of DeFi strategies from all the integrated cross-chain blockchains.

For the current revision, majority of the functionality is behind whitelisted and access controlled functions along with the cross-chain swapping process. There are also external strategy managers, and a part of divesting funds to strategies also happens through a manual process.

1.2   Methodology

During a security assessment, Zellic works through standard phases of security auditing including both automated testing and manual review. These processes can vary significantly per engagement, but the majority of the time is spent on a thorough manual review of the entire scope.

Alongside a variety of open-source tools and analyzers used on an as-needed basis, Zellic focuses primarily on the following classes of security and reliability issues:

Basic coding mistakes. Many critical vulnerabilities in the past have been caused by simple, surface-level mistakes that could have easily been caught ahead of time by code review. We analyze the scoped smart contract code using automated tools to quickly sieve out and catch these "shallow" bugs. Depending on the engagement, we may also employ sophisticated analyzers such as model checkers, theorem provers, fuzzers, etc. as necessary. We also perform a cursory review of the code to familiarize ourselves with the contracts.

Business logic errors. Business logic is the heart of any smart contract application. We manually review the contract logic to ensure that the code implements the expected functionality as specified in the platform's design documents. We also thoroughly examine the specifications and designs themselves for inconsistencies, flaws, and vulnerabilities. This involves use-cases that open the opportunity for abuse, such as flawed tokenomics or share pricing, arbitrage opportunities, etc.

Complex integration risks. Several high-profile exploits have been the result of not any bug within the contract itself, but rather an unintended consequence of its interaction with the broader DeFi ecosystem. We perform a meticulous review of all of the contract's possible external interactions, and summarize the associated risks; for

example: flash loan attacks, oracle price manipulation, MEV/sandwich attacks, etc.

Code maturity. We review for possible improvements in the codebase in general. We look for violations of industry best practices and guidelines, or code quality standards. We also provide suggestions for possible optimizations, such as gas optimization, upgradeability weaknesses, centralization risks, etc.

For each finding, Zellic assigns it an impact rating based on its severity and likelihood. There is no hard-and-fast formula for calculating a finding's impact; we assign it on a case-by-case basis based on our professional judgment and experience. As one would expect, both the severity and likelihood of an issue affect its impact; for instance, a highly severe issue's impact may be attenuated by a very low likelihood. We assign the following impact ratings (ordered by importance): Critical, High, Medium, Low, and Informational.

Similarly, Zellic organizes its reports such that the most important findings come first in the document, rather than impact alone. Thus, we may sometimes emphasize a "Informational" finding higher than a "Low" finding. The key distinction is that although certain findings may have the same impact rating, their importance may differ. This varies based on numerous soft factors, such as our clients' threat model, their business needs, project timelines, etc. We aim to provide useful and actionable advice to our partners that consider their long-term goals, rather than simply a list of security issues at present.

1.3 Scope

The engagement involved a review of the following targets:

f-smart-contracts

Repository	https://github.com/fractal-protocol/f-smart-contracts
Versions	ee0fa9e0fb04872c2349b90e1bcdce0dbcf98a99
Type	Solidity
Platform	Ethereum (and other compatible chains)

f-strategy-contracts

Repository	https://github.com/fractal-protocol/f-strategy-contracts
Versions	38761447a0379ee65946210b90590e0542d48bdb
Type	Solidity
Platform	Ethereum (and other compatible chains)

1.4   Project Overview

Zellic was approached to perform a two-week assessment with two consultants, for a total of 4 person-weeks.

Contact Information

The following project managers were associated with the engagement:

Jasraj Bedi, Co-founder
jazzy@zellic.io

Stephen Tong, Co-founder
stephen@zellic.io

The following consultants were engaged to conduct the assessment:

Jasraj Bedi, Co-founder
jazzy@zellic.io

Ayaz Mammadov, Senior Partner
ayaz@zellic.io

Project Timeline

The key dates of the engagement are detailed below.

March 4, 2022	Kick-off call
March 7, 2022	Start of primary review period
March 14, 2022	Weekly progress update
March 18, 2022	End of primary review period
March 22, 2022	Closing call

1.5   Disclaimer

This assessment does not provide any warranties on finding all possible issues within its scope; i.e., the evaluation results do not guarantee the absence of any subsequent issues. Zellic, of course, also cannot make guarantees on any additional code added to the assessed project after our assessment has concluded. Furthermore, because a single assessment can never be considered comprehensive, we always recommend multiple independent assessments paired with a bug bounty program. Finally, this assessment report should not be considered as financial or investment advice.

2 Executive Summary

Zellic conducted an audit for Fractal from March 4th to March 18th, 2022 on the scoped contracts and discovered 8 findings. Fortunately, no critical issues were found. We applaud Fractal for their attention to detail and diligence in maintaining high code quality standards. Of the 8 findings, 2 were of high impact, 1 was of medium impact, and 1 was of low impact. The remaining findings were informational in nature.

Fractal is a cross-chain yield farming aggregator, with the majority of current functionality locked behind centralized access-controlled methods. These methods would be called by thoroughly vetted, transparent, trusted third parties. Thus, for this audit, we focused heavily on the externally reachable attack surface, as bugs there would be of the highest impact.

Our general overview of the code is that it was very well organized and structured. The code coverage is high and tests are included for the majority of the functions. The documentation was adequate, although it could be improved. There were some pain points in a few areas that were confusing to read, but apart from those, the code was easy to comprehend.

Breakdown of Finding Impacts

Impact Level	Count
Critical	0
High	2
Medium	1
Low	1
Informational	4
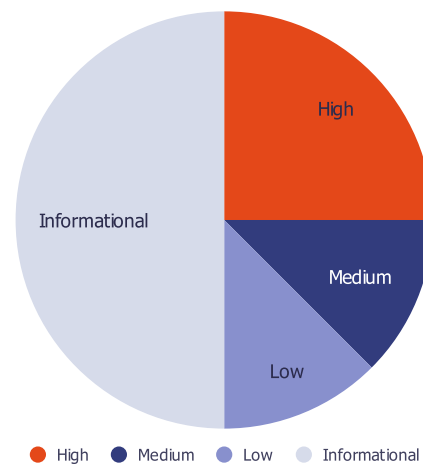
3 Detailed Findings

3.1 An attacker may claim risk-free rewards without risking their staked capital

- **Target**: Vault.sol
- **Severity**: High
- **Impact**: High

- **Category**: Business Logic
- **Likelihood**: High

Description

The Example Vault aims for an APR of 20%. At the beginning of every new period (1 day), the vault distributes the daily interest and calculates the new token price. The caveat here is that users can stake capital at the end of a period and reap rewards instantly at the beginning of the next period. Depositing on the last block before the start of new period and redeeming it in the next block would essentially guarantee an instant riskless profit.

```solidity
function compute () public {
    uint256 currentTimestamp = block.timestamp; // solhint-disable-line
    not-rely-on-time
    uint256 newPeriod = DateUtils.diffDays(startOfYearTimestamp,
    currentTimestamp);
    if (newPeriod <= currentPeriod) return;

    for (uint256 i = currentPeriod + 1; i <= newPeriod; i++) {
        _records[i].apr = _records[i - 1].apr;
        _records[i].totalDeposited = _records[i - 1].totalDeposited;

        uint256 diff = uint256(_records[i - 1].apr) *
        USDF_DECIMAL_MULTIPLIER * uint(100)/ uint256(365);
        _records[i].tokenPrice = _records[i - 1].tokenPrice + (diff /
        uint256(10000));
        _records[i].dailyInterest = _records[i - 1].totalDeposited *
        uint256(_records[i - 1].apr) / uint256(365) / uint256(100);
    }
    currentPeriod = newPeriod;
}
```

Impact

An attacker can effectively siphon out money from vaults without participating in the strategies or taking on any risk. The profit is directly dependent on attackers' capital. For a concrete example: With an APR of 20% and a capital of 1 Million USDC, the attacker can freely profit 540 dollars a day (0.054%) disregarding the gas fee. The profit scales linearly and for 10 million USDC, the profit would be $5400/day.

Recommendations

There are multiple strategies that can be taken to address this:

- Lock the users capital for a minimum period of time to prevent instant withdrawals.
- Immediately forward funds to the `yieldReserve`, so a large deposit is not withdrawable instantly.

Remediation

The issue has been acknowledged by Fractal.

3.2 Lack of slippage checks on DEX swaps

- **Target**: Multiple contracts
- **Severity**: High
- **Impact**: High
- **Category**: Business Logic
- **Likelihood**: High

Description

In many separate areas of the project, interactions and swaps with Uniswap are handled through DexLibrary. There is no slippage check on these interactions and are thus vulnerable to market manipulation.

```solidity
function swap(
    uint256 amountIn,
    address fromToken,
    address toToken,
    IPair pair
) internal returns (uint256) {
    (address token0, ) = sortTokens(fromToken, toToken);
    (uint112 reserve0, uint112 reserve1, ) = pair.getReserves();
    if (token0 != fromToken) (reserve0, reserve1) = (reserve1, reserve0);
    uint256 amountOut1 = 0;
    uint256 amountOut2 = getAmountOut(amountIn, reserve0, reserve1);
    if (token0 != fromToken)
        (amountOut1, amountOut2) = (amountOut2, amountOut1);
    safeTransfer(fromToken, address(pair), amountIn);
    pair.swap(amountOut1, amountOut2, address(this), ZERO_BYTES);
    return amountOut2 > amountOut1 ? amountOut2 : amountOut1;
}
```

Impact

Due the nature of most of the vulnerable methods being `onlyOwner` or `onlyAdmin`, the quantity of funds accumulated would be rather large along with the swap amount. An attacker could sandwich the the swap transaction, artificially inflating the spot price and profiting off the manipulated market conditions when the swap executes.

Recommendations

Set the default slippage to 0.5% for Uniswap, customizable for bigger trades.

Remediation

The issue has been acknowledged by Fractal.

3.3 Potential lock-up of funds in FractalVaultV1 as anySwap Router is not approved

- **Target**: FractalVaultV1.sol
- **Severity**: Medium
- **Impact**: **Medium**

- **Category**: Business Logic
- **Likelihood**: Medium

Description

The FractalVaultV1 does not approve the anySwap router before executing `anySwapOut-Underlying`, and would fail all the withdrawal attempts.

```
function withdrawToLayerOne( ... ) {
        ...
        emit WithdrawToLayerOne(msg.sender, amount);

        anySwapRouter.anySwapOutUnderlying(anyToken, anyswapRouter,
    amount, chainId);
    }
```

Impact

The FractalVaultV1 will never be able to withdraw to LayerOne. Though the `recoverERC20` function can be used in an emergency to manually transfer funds as a backup functionality; however, this is likely not the intended flow of funds.

Recommendations

Approve AnySwap router before `anySwapOutUnderlying`.

Remediation

The issue has been acknowledged by Fractal.

3.4 Potential lock-up of funds in the event of insufficient AnySwap liquidity

- **Target**: FractVaultV1.sol
- **Severity**: Low
- **Impact**: Low

- **Category**: Business Logic
- **Likelihood**: Low

Description

AnySwap cross-chain transfers will provide the underlying token to the destination only if sufficient liquidity exists on AnySwap reserves. If not, AnySwap will mint a wrapped token (AnyToken) that can be redeemed later when liquidity is available. The FractVaultV1 does not handle that. Even if reserves are checked before executing a swap, since AnySwap is not atomic with no guarantee on order of transactions, simultaneous swaps by other users would lead to locked tokens.

Impact

FractalVaultV1 currently has no way to redeem the AnyTokens to the underlying tokens. However, the `recoverERC20` method can be used by the owner to manually recover the anySwap tokens, mitigating this issue's impact.

Recommendations

Add functionality to redeem AnyTokens to their underlying.

Remediation

The issue has been acknowledged by Fractal.

3.5 Access Control functions should emit events

- **Target**: Mintable.sol, Address-Whitelist.sol, Migrations.sol
- **Severity**: Informational
- **Impact**: Informational
- **Category**: Access Control
- **Likelihood**: N/A

Description

Several methods in multiple contracts related to access control such as whitelisting and minter/burner roles do not emit events.

Impact

In the case of a compromise, events allow for secure and early detection of breaches & security incidents.

Recommendations

Add events to all functions relating to access control.

Remediation

The issue has been acknowledged by Fractal.

3.6 Multiple internal inconsistencies

- **Target**: Multiple contracts
- **Severity**: Informational
- **Impact**: Informational
- **Category**: Business Logic
- **Likelihood**: N/A

Description

In several areas of the project, internal inconsistencies were noted, such as lack of checks that were present in other areas, or non-standard practices in general.

The respective areas are affected:

- FractalVaultV1: `withdrawToLayerOne` – No chainId Checks.
- Mintable.sol: `mint` – Transfer event should mint from address 0.
- DexLibrary.sol: `convertRewardTokensToDepositTokens` – lack of slippage checks mentioned.

Impact

These issues are minor, and do not pose a security hazard at present. More broadly however, this is a source of developer confusion and a general coding hazard. Internal inconsistencies may lead to future problems or bugs. Avoiding internal inconsistencies also makes it easier for developers to understand the code and helps any potential auditors more quickly and thoroughly assess it.

Recommendations

Consider changing the code to fix the inconsistencies.

Remediation

The issue has been acknowledged by Fractal.

3.7 Lack of documentation

- **Target**: Multiple contracts
- **Severity**: Informational
- **Impact**: Informational
- **Category**: Business Logic
- **Likelihood**: N/A

Description

Several files in the project are lacking documentation, the following being:

- DateUtils.sol: `diffDays`
- DateUtils.sol: `_daysToDate`
- DateUtils.sol: `_daysFromDate`
- DateUtils.sol: `getYear`
- DateUtils.sol: `timestamp`
- Migrations.sol: `setCompleted`

Impact

This is a source of developer confusion and a general coding hazard. Lack of documentation, or unclear documentation, is a major pathway to future bugs. It is best practice to document all code. Documentation also helps third-party developers integrate with the platform, and helps any potential auditors more quickly and thoroughly assess the code.

Recommendations

Add documentation to the affected functions.

Remediation

The issue has been acknowledged by Fractal.

3.8 Insufficient code documentation

- **Target**: DateUtils.sol
- **Severity**: Informational
- **Impact**: Informational
- **Category**: Business Logic
- **Likelihood**: N/A

Description

We found that the code quality unsatisfactory for certain functions, namely:

- DateUtils.sol: _daysToDate

```solidity
function _daysToDate(uint256 _days) internal pure returns (uint256
    year, uint256 month, uint256 day) {
    int256 __days = int256(_days);

    int256 L = __days + 68569 + OFFSET19700101;
    int256 N = 4 * L / 146097;
    L = L - (146097 * N + 3) / 4;
    int256 _year = 4000 * (L + 1) / 1461001;
    L = L - 1461 * _year / 4 + 31;
    int256 _month = 80 * L / 2447;
    int256 _day = L - 2447 * _month / 80;
    L = _month / 11;
    _month = _month + 2 - 12 * L;
    _year = 100 * (N - 49) + _year + L;
        ...
```

_daysToDate uses a lot of abstract math to converts days to a date.

- DateUtils.sol: _daysFromDate

```solidity
function _daysFromDate(uint256 year, uint256 month, uint256 day)
    internal pure returns (uint256 _days) {
    require(year >= 1970, "Error");
    int _year = int(year);
    int _month = int(month);
    int _day = int(day);

    int __days = _day
        - 32075
        + 1461 * (_year + 4800 + (_month - 14) / 12) / 4
        + 367 * (_month - 2 - (_month - 14) / 12 * 12) / 12
```

```
        - 3 * ((_year + 4900 + (_month - 14) / 12) / 100) / 4
        - OFFSET19700101;

    _days = uint256(__days);
    }
```

_daysFromDate uses a lot of optimized math to converts days to a date.

- Vault.sol: `Compute`

```
function compute () public {
        ...
        for (uint256 i = currentPeriod + 1; i <= newPeriod; i++) {
            _records[i].apr = _records[i - 1].apr;
            _records[i].totalDeposited = _records[i - 1].
    totalDeposited;

            uint256 diff = uint256(_records[i - 1].apr) *
    USDF_DECIMAL_MULTIPLIER * uint256(100) / uint256(365);
            _records[i].tokenPrice = _records[i - 1].tokenPrice + (
    diff / uint256(10000));
            _records[i].dailyInterest = _records[i - 1].
    totalDeposited * uint256(_records[i - 1].apr) / uint256(365) /
    uint256(100);
        }
        ...
    }
```

A lack of comments here renders this function difficult to understand.

Impact

Code maturity is very important in a code base, this is because commented out code and unused variables can result in increased complexity and confusion when developers have to modify the business logic.

Remediation

The issue has been acknowledged by Fractal.

4 Discussion

In this section, we discuss miscellaneous interesting observations during the audit that are noteworthy and merit some consideration.

We applaud Fractal's initiative for taking on the challenge of yield farming over fragmented liquidity in multiple chains.

The quality of code is commendable, and the test coverage reaches almost 100% (99.6%).

Another point to consider may be that many functions such as `mint` which are allowed to be called by whitelisted addresses dictated by the owner pose a large centralization risk. This is by design but we would still suggest the following:

- Use a multi-signature address wallet, this would prevent an attacker from causing irreversible damage if the EOA wallet were compromised.
- Place dangerous functions like whitelists behind a timelock to catch malicious executions in the case of compromise.