Zellic

Volt Protocol

Smart Contract Security Assessment

March 24, 2022

Prepared for:

Elliot Friedman

Volt Protocol

Prepared by:

Stephen Tong and Chad McDonald

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 Volt Protocol

Volt is a stablecoin that aims to preserve purchasing power and mitigate the effects of inflation. It does this by tracking the Consumer Price Index (CPI).

Volt is part of the Fei Protocol family of projects, and shares a significant part of its codebase with Fei. We were approached to audit a component in Volt, the ScalingPriceOracle. The ScalingPriceOracle essentially is a chainlink client which brings CPI data on-chain through a chainlink oracle. We also reviewed the new global rate limited minter and non-custodial PSM, which were largely unchanged since our previous audit of Fei Core.

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:

volt-protocol-core

Repository	https://github.com/volt-protocol/volt-protocol-core
Versions	`eaf40a779cac3cb8742c3ebfd1b8bbda411c986a`
Type	Solidity
Platform	Ethereum

Contracts:

- GlobalRateLimitedMinter
- MultiRateLimited
- Deviation
- ScalingPriceOracle
- OraclePassThrough
- NonCustodialPSM

1.4 Project Overview

Zellic was approached to perform a 3-day assessment with two consultants, for a total of 1 person-week.

Contact Information

The following project managers were associated with the engagement:

Stephen Tong, Co-founder
stephen@zellic.io

The following consultants were engaged to conduct the assessment:

Stephen Tong, Co-founder
stephen@zellic.io

Chad McDonald, Consultant
chad@zellic.io

Project Timeline

The key dates of the engagement are detailed below.

March 7, 2022	Kick-off call
March 21, 2022	Start of primary review period
March 24, 2022	End of primary review period
TBD	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 Volt Protocol from March 21st to March 23rd, 2022 on the scoped contracts and discovered 4 findings. Fortunately, no critical issues were found. Of the 4 findings, all were informational in nature. We applaud Fei for their attention to detail and diligence in maintaining high code quality standards.

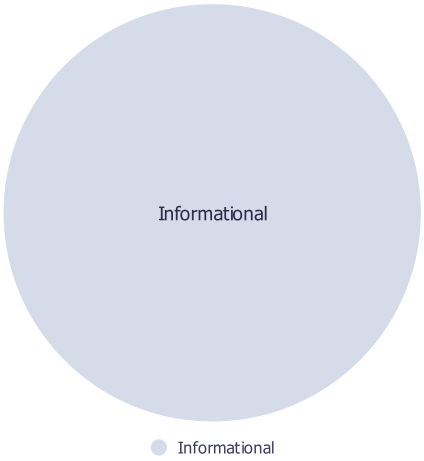
Volt is a stablecoin that is designed to maintain purchasing power, by tracking the Consumer Price Index (CPI). This data is fed on-chain through a chainlink oracle into the ScalingPriceOracle. This contract effectively implements a 1-month linear interpolation to smoothly change value between one month and the next. During our review, we found that it correctly uses and implements the chainlink client in a fashion consistent with Chainlink's first-party documentation.

Much of the project's code base has already been audited previously. We were approached to audit the ScalingPriceOracle and its auxiliary code. We also reviewed the new global rate limited minter and non-custodial PSM, which were largely unchanged since our previous audit of Fei Core. We were instructed to leave the contracts and code outside the scope of this assessment.

Our general overview of the code is that it is very well-written and maintained. Within the assessment scope, the test suite reaches nearly 100% code coverage, and the project has adopted continuous integration. The documentation was clear, concise, and thorough. We hope Volt continues the commitment to high code quality that they have shown so far.

Breakdown of Finding Impacts

Impact Level	Count
Critical	0
High	0
Medium	0
Low	0
Informational	4



3 Detailed Findings

3.1 Missing test suite code coverage

- **Target**: MultiRateLimited
- **Severity**: Low
- **Impact**: Informational
- **Category**: Code Maturity
- **Likelihood**: n/a

Description

Some functions in the smart contract are not covered by any unit or integration tests, to the best of our knowledge. We ran both the Hardhat test suite and the Forge tests. The following functions do not have test coverage:

MultiRateLimited.sol: `getLastBufferUsedTime`

These functions are extremely simple, so we do not see this as a significant issue.

We reviewed all untested functions with increased scrutiny. Fortunately, we did not find any additional vulnerabilities.

Other than these minor flaws, the code base otherwise has nearly 100% code coverage as of the time of writing. We applaud Volt Protocol for their commitment to thorough testing.

Impact

Because correctness is so critically important when developing smart contracts, we recommend that all projects strive for 100% code coverage. Testing should be an essential part of the software development lifecycle. No matter how simple a function may be, untested code is always prone to bugs.

Recommendations

Expand the test suite so that all functions and their branches are covered by unit or integration tests.

Remediation

The issue has been acknowledged by Volt Protocol, and a fix is pending.

3.2 Time functions rely on an unverified external date library

- **Target**: ScalingPriceOracle
- **Severity**: n/a
- **Impact**: Informational
- **Category**: Business Logic
- **Likelihood**: n/a

Description

The ScalingPriceOracle is designed so that new CPI data from the chainlink oracle can be requested once a month: at least 28 days between requests, and only on the 15th day of the month or later. To calculate the current day, the contract uses `block.timestamp` and the popular BokkyPooBah's DateTime Library. This library contains an algorithm to convert from Unix timestamp days since epoch to the current calendar date.

```
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;
```

Since this code is crucial to the functionality of the contract, and its design is not clearly documented, we considered the risk of a possible bug in this dependency.

A bug in the dependency could cause the ScalingPriceOracle to malfunction or lock up. This is mitigated by the fact that the ScalingPriceOracle is kept behind a proxy)OraclePassThrough), but we still wanted to verify the correctness of this function.

To do so, we compared the results of the BokkyPooBah algorithm with a known ground truth)Python's `datetime` library). We computed values with both methods for all timestamp values +/- 30 years from the current date, and found that the results were all correct.

We also formally verified the correctness of the algorithm against glibc's `gmtime` function by using an SMT solver.

Recommendations

In the future, continue to carefully verify the correctness of external dependencies when adding them to the code base. There have been several well-known security incidents caused by external dependencies in the past.

Remediation

No remediation is necessary, as we successfully verified the dependencies are correct.

3.3 Some functions can be implemented more efficiently

- **Target**: Deviation
- **Severity**: Low
- **Impact**: Informational
- **Category**: Gas Optimization
- **Likelihood**: n/a

Description

The function `calculateDeviationThresholdBasisPoints` calculates an absolute difference of two numbers. It computes the absolute value of both the numerator and denominator separately, which is less efficient than computing the absolute value of the overall result.

```
function calculateDeviationThresholdBasisPoints(int256 a, int256 b)
    public pure returns (uint256)
{
    /// delta can only be positive
    uint256 delta = ((a < b) ? (b - a) : (a - b)).toUint256();

    return
        (delta * Constants.BASIS_POINTS_GRANULARITY) /
        (a < 0 ? a * -1 : a).toUint256();
}
```

Recommendations

The code can be refactored to compute the absolute value of the quotient at the end, rather than computing the quotient of two absolute values. This would eliminate a ternary expression.

Remediation

Volt Protocol acknowledged and optimized the code based on our suggestions.

3.4 Some variable names do not follow naming conventions

- **Target**: IScalingPriceOracle
- **Severity**: n/a
- **Impact**: Informational
- **Category**: Code Maturity
- **Likelihood**: n/a

Description

The functions `TIMEFRAME` and `MAXORACLEDEVIATION` do not follow Solidity's mixedCase naming convention for functions.

Impact

Code comprehension may be diminished, possibly hindering developers' ability to easily read and understand the code.

Recommendations

We recommend renaming these functions using camelCase. The intention behind capitalizing `TIMEFRAME` and `MAXORACLEDEVIATION` is to distinguish them as an important functions so consider highlighting their importance in a code comment.

Remediation

The issue has been acknowledged by Volt Protocol.

4 Discussion

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

Volt's test suite is exceptionally thorough. Across the core code base, there is high test coverage, with both unit and integration tests. They have also adopted Forge and plan to add fuzz tests in the future. Lastly, Volt has adopted **continuous testing** as part of their development process; tests are run by CI for all merge requests. We applaud Volt for their commitment to thorough testing, and recommend other projects to adopt similar practices.

It is important to note that 100% branch coverage is not a security guarantee. Even if all lines of code are covered by tests, the state space of all possible variables values is not necessarily covered. For this reason, we also recommend adopting fuzz testing to cover more of this aforementioned variable state space. Notwithstanding that, not even fuzz testing is infallible. Only formal verification would approach the level assurance needed for such a guarantee. Therefore, we also recommend that projects explore formal verification of their Solidity code.