Zellic

Valorem Options

Smart Contract Security Assessment

April 10, 2023

Prepared for:

Alcibiades

Valorem Labs Inc

Prepared by:

Katerina Belotskaia, Vlad Toie, Junyi Wang

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 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 contact us at hello@zellic.io.

1 Executive Summary

Zellic conducted a security assessment for Valorem Labs Inc from January 11th to January 13th, 2023. During this engagement, Zellic reviewed Valorem Options's code for security vulnerabilities, design issues, and general weaknesses in security posture.

1.1 Goals of the Assessment

In a security assessment, goals are framed in terms of questions that we wish to answer. These questions are agreed upon through close communication between Zellic and the client. In this assessment, we sought to answer the following questions:

- How does Valorem assign options and claims fairly?
- How are buckets organized and how are options assigned to buckets?
- Is there any way to trick the protocol into preferentially exercising options from a specific bucket?
- Are there any circumstances when an arbitrager could profit from a certain option and/or market conditions?
- Are users always able to redeem their options for the underlying asset at expiry/exercise?
- Are any of the imposed restrictions bypassable?

1.2 Non-goals and Limitations

- Problems related to liquidity of either underlying or exercising assets or any unexpected market behavior
- Problems relating to the front-end components and infrastructure of the project
- Issues stemming from code or infrastructure outside of the assessment scope

During the assessment, we prioritized the fair assignment of claims and options, which limited the scope of our investigation into potential economic attacks on the protocol. While we have confidence in the security of the protocol, we cannot offer a definitive assurance that it is invulnerable to economic attacks.

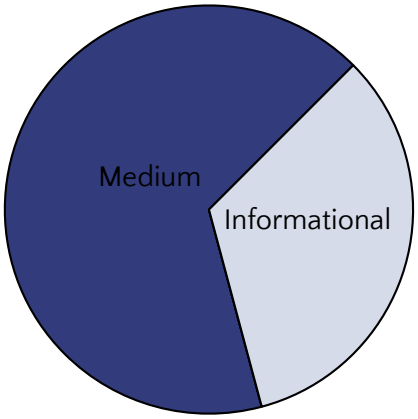
It is important to note that the purpose of this assessment is to verify the effectiveness of the measures implemented by Valorem to address issues identified in our previous assessment. For that reason, we have prioritized the verification of the mitigations over the discovery of new issues.

1.3 Results

During our assessment on the scoped Valorem Options contracts, we discovered three findings. No critical issues were found. Two findings were of medium impact and one finding was informational.

Breakdown of Finding Impacts

Impact Level	Count
Critical	0
High	0
Medium	2
Low	0
Informational	1



2 Introduction

2.1 About Valorem Options

Valorem helps DeFi power users pursue advanced hedging strategies by writing their own physically settled options permissionlessly on any ERC20 token.

2.2 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 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. Depending on the engagement, we may also employ sophisticated analyzers such as model checkers, theorem provers, fuzzers, and so on 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 examine the specifications and designs for inconsistencies, flaws, and weaknesses that create opportunities for abuse. For example, these include problems like unrealistic tokenomics or dangerous arbitrage opportunities. To the best of our abilities, time permitting, we also review the contract logic to ensure that the code implements the expected functionality as specified in the platform's design documents.

Integration risks. Several well-known exploits have not been the result of any bug within the contract itself; rather, they are an unintended consequence of the contract's interaction with the broader DeFi ecosystem. Time permitting, we review the contracts' external interactions and summarize the associated risks: for example, flash loan attacks, oracle price manipulation, MEV/sandwich attacks, and so on.

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

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. Instead, we assign it on a case-by-case basis based on our judgment and experience. Both the severity and likelihood of an issue affect its impact. For instance, a highly severe issue's impact may be attenuated by a low likelihood. We assign the following impact ratings (ordered by importance): Critical, High, Medium, Low, and Informational.

Zellic organizes its reports such that the most important findings come first in the document, rather than being strictly ordered on impact alone. Thus, we may sometimes emphasize an "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 various soft factors, like our clients' threat models, their business needs, and so on. We aim to provide useful and actionable advice to our partners considering their long-term goals, rather than a simple list of security issues at present.

2.3 Scope

The engagement involved a review of the following targets:

Valorem Options Contracts

Repository	https://github.com/valorem-labs-inc/valorem-core/
Versions	`ed53af23ff17d2c04aaa25f1b038d0a4081288b7`
Programs	• OptionSettlementEngine • TokenURIGenerator
Type	Solidity
Platform	EVM-compatible

2.4 Project Overview

Zellic was contracted to perform a security assessment with three consultants for a total of one person-week. The assessment was conducted over the course of three calendar days.

Contact Information

The following project managers were associated with the engagement:

Chad McDonald, Engagement Manager
chad@zellic.io

The following consultants were engaged to conduct the assessment:

Junyi Wang, Engineer
junyi@zellic.io

Katerina Belotskaia, Engineer
kate@zellic.io

Vlad Toie, Engineer
vlad@zellic.io

2.5 Project Timeline

The key dates of the engagement are detailed below.

January 11, 2023	Start of primary review period
January 13, 2023	End of primary review period

3 Detailed Findings

3.1 Bucket for `exercise()` manipulatable with small `exercise()` calls

- **Target**: OptionSettlementEngine
- **Category**: Business Logic
- **Likelihood**: Medium
- **Severity**: High
- **Impact**: Medium

Description

The first bucket to be exercised in an `exercise()` call is based on a deterministic seed. This seed is reset upon every `exercise()` call. The seed can be intentionally reset by `exercise()`ing with a small amount. If the seed is repeatedly reset until the desired bucket is next in line, the next bucket to be exercised can effectively be chosen this way.

The code to choose buckets and exercise are as follows:

```
function _assignExercise(OptionTypeState storage optionTypeState,
    Option storage optionRecord, uint112 amount)
        private
    {
        // Setup pointers to buckets and buckets with collateral available
    for exercise.
        // ...
        uint96 numUnexercisedBuckets
    = uint96(unexercisedBucketIndices.length);
        uint96 exerciseIndex = uint96(optionRecord.settlementSeed %
    numUnexercisedBuckets);

        while (amount > 0) {
            // ...

            if (amount ≠ 0) {
                exerciseIndex = (exerciseIndex + 1) %
    numUnexercisedBuckets;
            }
        }
```

```
        // Update the seed for the next exercise.
        optionRecord.settlementSeed =

    uint160(uint256(keccak256(abi.encode(optionRecord.settlementSeed,
    exerciseIndex))));
    }
```

Impact

A user who owns options can effectively choose the next bucket to be exercised for that option category. This can be used to reduce the exercise priority of one's own options or force some specific bucket of options to be preferentially exercised.

Recommendations

Reset the `settlementSeed` only if at least one bucket is exhausted.

Remediation

The `settlementSeed` is now only randomized for the first bucket. It was fixed in commit 1d6c08b43.

3.2 Probability of bucket `exercise()` not correlated with size

- **Target**: OptionSettlementEngine
- **Category**: Business Logic
- **Likelihood**: N/A
- **Severity**: Medium
- **Impact**: Medium

Description

The probability of an options bucket being chosen for `exercise()` is not correlated with the number of options contained in that bucket. Individual options in smaller buckets have a higher probability of being chosen for exercise.

The first bucket to be exercised per `exercise()` call is chosen pseudorandomly with uniform probability for all buckets as follows:

```
uint96 numUnexercisedBuckets = uint96(unexercisedBucketIndices.length);
        uint96 exerciseIndex = uint96(optionRecord.settlementSeed %
    numUnexercisedBuckets);
```

Impact

Since the probability of exercise is not normalized by bucket size, options in smaller buckets have a higher expected amount exercised per option. If writing a small amount of options, this can be disadvantageous if unable to write into a larger bucket.

Recommendations

Base the probability of a bucket being chosen on the size of the bucket or some other criterion to ensure fairness.

Remediation

Since the commit 1d6c08b43, only the first bucket to be exercised is random. Since the number of randomizations is now vastly reduced, the bias of the uneven randomization has been drastically improved. Although the choice of first bucket is still biased.

Valorem Labs Inc plans to fully remediate this in a future version.

3.3 Bucket for `exercise()` is known in advance to users

- **Target**: OptionSettlementEngine
- **Category**: Business Logic
- **Likelihood**: Informational
- **Severity**: Informational
- **Impact**: Informational

Description

Users can determine the probability of a bucket being exercised within a specific time interval using a statistical model with some degree of accuracy.

Impact

A user may base the decision to write options or not on the predicted sequence of exercised buckets. This could be a minor advantage for certain users of option pools with overlapping write and exercise intervals.

Remediation

Valorem Labs Inc has acknowledged this behavior and have plans to remediate this in the future.

4 Discussion

The purpose of this section is to document miscellaneous observations that we made during the assessment.

4.1 Rounding error in the `redeem` mechanism

The rounding error in the `redeem` mechanism problem was discovered during the first audit phase, and in order to address the issue, Valorem implemented significant changes to the options' writing mechanism and overall contract architecture. Therefore, a thorough examination was conducted during the current audit phase to confirm that these changes have effectively resolved the issue.

Remediation

The issue has been fixed in commit 18b2a9e2 by performing all necessary multiplication operations before performing the division to calculate the amount of exercised and unexercised options.

4.2 Writing during the exercise period may lead to arbitrage opportunities

The writing during the exercise period may lead to arbitrage opportunities problem was discovered during the first audit phase, and in order to address the issue, Valorem implemented significant changes to the options' writing mechanism and overall contract architecture. Therefore, a thorough examination was conducted during the current audit phase to confirm that these changes have effectively resolved the issue.

Remediation

The issue has been fixed in commit 18b2a9e2 by prohibiting writing to the current bucket if it was previously exercised and creating a new bucket to write this option.

4.3 The `claimRecord.amountWritten` is not reset in the `redeem()` function

The `claimRecord.amountWritten` is not reset in the `redeem()` function problem was discovered during the first audit phase, and in order to address the issue, Valorem im-

plemented significant changes to the options' writing mechanism and overall contract architecture. Therefore, a thorough examination was conducted during the current audit phase to confirm that these changes have effectively resolved the issue.

Remediation

The issue has been fixed in commit e1f0f121 by changing the design of storing information about claims and removing internal `_claim` mapping.

4.4 The `_claimIdToClaimIndexArray` mapping is not reset in the `redeem()` function

The `_claimIdToClaimIndexArray` mapping is not reset in the `redeem()` function problem was discovered during the first audit phase, and in order to address the issue, Valorem implemented significant changes to the options' writing mechanism and overall contract architecture. Therefore, a thorough examination was conducted during the current audit phase to confirm that these changes have effectively resolved the issue.

Remediation

The issue has been fixed in commit e1f0f121 by changing the design of storing information about claims and deleting information about each `claimIndices` during the calculation of exercised and unexercised amounts of options in the `redeem` function.

4.5 Invariants for testing

During the course of the audit we noted invariants that may be useful when writing tests.

1. The total value held by the contract should only increase by the amount written when an option is written and decrease by the amount redeemed when an option is redeemed. Otherwise, it should not change. Note that the exchange rate of exercise to underlying token is determined by the option.
2. The sum of exercised and unexercised buckets should be identical to the contract balance. This ensures that buckets are exercisable after accounting for fees and any possible rounding errors.
3. The exercised amount of assets for buckets should not be more than the written amount.

4.6 Off-chain claim tracking

During the audit we recommended that the OptionSettlementEngine contract track the list of claims for a user off-chain. Since there can be only one claim per bucket for every user, the number of claims is still limited by the number of buckets and hence O(sqrt(n)). When the user is writing, they can pass in an existing `ClaimIndex` to write to that claim (with appropriate checks, of course), instead of creating a new claim.

While Valorem Labs Inc did not adopt all these suggestions, they did add events to facilitate off-chain tracking of claims as they currently are.

5 Threat Model

This provides a full threat model description for various functions. As time permitted, we analyzed each function in the smart contracts and created a written threat model for some critical functions. A threat model documents a given function's externally controllable inputs and how an attacker could leverage each input to cause harm. Not all functions in the audit scope may have been modeled. The absence of a threat model in this section does not necessarily suggest that a function is safe.

5.1 File: OptionSettlementEngine

Function `claim()`:

Intended behavior

- Provides information about the claim of a given `claimId`.
- Returns an approximation of the amount of underlying and exercise assets that can be claimed for a given `claimId`.

Branches and code coverage

Intended branches:

- Assumes that `decodeTokenId` call works fine (the returned values are equal to the expected).
  ☑ Test coverage
- Ensures that the claim it has searched for actually exists.
  ☑ Test coverage

Negative behavior:

- Will revert if `claimId` is not initialized.
  ☑ Negative test?

Preconditions

- Assumes that the claim actually exists.

Inputs

- `claimId`:
  - **Control**: Full control.

- **Authorization**: Checked whether it exists as well if the option attributed to it has been initialized or not.
- **Impact**: Allows to get values for any `claimId`. Can be used for getting information about which bucket was exercised, if `claimId` connected with only one bucket.

Function: `position()`

Intended behavior

- Provides information about the position underlying a token - useful for determining the value.
- Returns total amount of underlying associated with given `optionKey` for `optionId` or `claimId`.

Branches and code coverage

Intended branches:

- Assumes that `decodeTokenId` call works fine.
  ☑ Test coverage
- Ensures that the token it has searched for actually exists.
  ☑ Test coverage
- Ensures that the return values for not exercised `claimId` are expected (exercisePosition == 0; underlyingPosition == amount * underlyingAmount).
  ☑ Test coverage
- Ensures that the return values after exercise are expected.
  ☑ Test coverage

Negative behavior:

- Will revert if `tokenId` is not initialized.
  ☑ Negative test?

Preconditions

- Assumes that the position actually exists.
- Assumes that the `tokenId` is either a `claimId` or `optionId`.

Inputs

- `tokenId`:
  - **Control**: Full control.
  - **Authorization**: checked whether it exists as well as if the option attributed

to it has been initialized or not.
  - **Impact**: Allows to get values for any `tokenId`. Can be used for getting information about which bucket was exercised, if `claimId` connected with only one bucket.

Function call analysis

- `decodeTokenId(tokenId)`
  - **What is controllable?** `tokenId` controlled.
  - **If return value controllable, how is it used and how can it go wrong?**
- Used to determine the `optionKEY` and the `claimNum` attributed to a certain `tokenId`.
  - Would be bad if two `tokenIds` point to the same thing.
- **What happens if it reverts, reenters, or does other unusual control flow?** No problems.

- `isOptionInitialized(optionKey)`
  - **What is controllable?** The `optionKey` that is used through the specified `tokenId`.
  - **If return value controllable, how is it used and how can it go wrong?** Returns whether an `option` exists at that `optionKey`.
  - **What happens if it reverts, reenters, or does other unusual control flow?** It means `option` doesn't exist for that key.
- `_getAssetAmountsForClaimIndex(underlyingAssetAmount, exerciseAssetAmount, optionTypeState, claimIndices, i)`
  - **What is controllable?** The `tokenId` that's used to get the `optionKey` and `claimNum` associated with that particular claim or option.
  - **If return value controllable, how is it used and how can it go wrong?** The return values depend on how many tokens were exercised from the bucket with `claimId` tokens and how many tokens were written to the current bucket. The wrong value can be returned if there are calculation mistakes.
  - **What happens if it reverts, reenters, or does other unusual control flow?** N/A.

Function: `_encodeTokenId()`

Intended behavior

Pure function. Determines the `tokenId` value for given `optionKey` and `claimNum`.

Branches and code coverage

Intended branches:

- Ensures that the return values are calculated correctly (first 20 bytes is `optionKey` and the last 12 bytes is `claimNum`).
  ☑ Test coverage

Inputs

- `optionKey`:
  - **Control**: Full control.
  - **Authorization**: No checks.
  - **Impact**: No impact.
- `claimNum`:
  - **Control**: Full control.
  - **Authorization**: No checks.
  - **Impact**: No impact.

Function call analysis

There aren't external calls here.

Function: `_decodeTokenId()`

Intended behavior

Pure function. Allows to parse from the `tokenId` the `optionKey` and `claimNum` values.

Branches and code coverage

Intended branches:

- Ensures that the return values are calculated correctly (first 20 bytes from `tokenId` is `optionKey` and the last 12 bytes is `claimNum`).
  ☑ Test coverage

Inputs

- `tokenId`:
  - **Control**: Full control.
  - **Authorization**: No checks.
  - **Impact**: No impact.

Function call analysis

There aren't external calls here.

Function: `newOptionType()`

Intended behavior

- Should create a new `option` if one doesn't already exist for the given parameters.

Branches and code coverage

Intended branches:

- Check that `exerciseTimestamp` is at least 24h after `expiryTimestamp` (as per documentation).
  - ☑ Test coverage
- Check that `expiryTimestamp` > 24h.
  - ☑ Test coverage
- Check that `exerciseTimestamp` and `expiryTimestamp` are both in the future.
  - ☑ Test coverage
- Check that the `underlyingAsset` and `exerciseAsset` are different.
  - ☑ Test coverage
- Check that `expiryTimestamp` is not 0. This cannot happen since `expiryTimestamp` already has to be greater than the `block.timestamp` by at least 24h.
  - ☑ Test coverage

Negative behavior:

- Make sure `underlyingAsset` ≠ `exerciseAsset`.
  - ☑ Negative test? Covered by `if-case`
- Revert if `expiryTimestamp` < 24h
  - ☑ Negative test?
- Revert if exercise window is less than 24h
  - ☑ Negative test?
- Revert if `optionKey` already exists
  - ☑ Negative test? Covered by `isOptionInitialized(optionKey)`

Preconditions

- Assumes that a pair with literally the same details doesn't already exist.
- Assumes that the encoding used to calculate the `optionKey` is unique and that it cannot be duplicated.

Inputs

- `underlyingAsset`:
  - **Control**: Only existing contract address.
  - **Authorization**: underlyingToken.totalSupply() >= underlyingAmount.
  - **Impact**: N/A.
- `underlyingAmount`:
  - **Control**: Limited control.
  - **Authorization**: UnderlyingToken.totalSupply() >= underlyingAmount.
  - **Impact**: N/A.
- `exerciseAsset`:
  - **Control**: Only existing contract address.
  - **Authorization**: exerciseToken.totalSupply() >= exerciseAmount.
  - **Impact**: N/A.
- `exerciseAmount`:
  - **Control**: Limited control.
  - **Authorization**: exerciseToken.totalSupply() >= exerciseAmount.
  - **Impact**: N/A.
- `expiryTimestamp`:
  - **Control**: Full control.
  - **Authorization**: expiryTimestamp >= (block.timestamp + 1 day).
  - **Impact**: If `exerciseTimestamp` is in the distant future, it will not be possible to call the `redeem` function.
- `exerciseTimestamp`:
  - **Control**: Full control.
  - **Authorization**: expiryTimestamp >= (exerciseTimestamp + 1 day).
  - **Impact**: It is possible to call the `write` and the `exercise` functions the same time.

Function call analysis

- `isOptionInitialized(optionKey)`
  - **What is controllable?** To some extent, the `optionKey` is controllable, since it has just been generated.
  - **If return value controllable, how is it used and how can it go wrong?** Return value is used to determine whether the specific `optionKey` has been initialized.
  - **What happens if it reverts, reenters, or does other unusual control flow?** Cannot revert; it can return `false` though when the `optionKey` was not initialized.

- `underlyingToken.totalSupply()` and `exerciseToken.totalSupply()`
  - **What is controllable?** The `underlyingToken` and `exerciseToken` are both arbitrary.
  - **If return value controllable, how is it used and how can it go wrong?** Since both tokens are arbitrary, their `totalSupply` can be manipulated; it's still a good check since you wouldn't want erroneous tokens like that in your app.
  - **What happens if it reverts, reenters, or does other unusual control flow?** Basically need to assess how dangerous supplying arbitrary tokens is.

Function: `write()`

Intended behavior

- Write an amount of a specified `option`. If there's no claim for the `option` yet, create one. If there is, `write` more options into it.

Branches and code coverage

Intended branches:

- Check `option` expiration.
  ☑ Test coverage
- Check `option` exists.
  ☑ Test coverage
- If `msg.sender` has enough tokens and `optionId` is valid, they receive the expected amount of `optionId` and one claim NFT.
  ☑ Test coverage
- Should be called successfully before `expiryTimestamp`
  ☑ Test coverage

Negative behavior:

- Should revert if `msg.sender` doesn't have enough `underlyingAsset` tokens.
  ☑ Negative test?
- Should revert if `msg.sender` doesn't own `claimId` token.
  ☑ Negative test?
- Should revert if `optionId` is invalid (the last 12 bytes are not zero).
  ☑ Negative test?
- Should revert if `optionId` does not exist.
  ☑ Negative test?
- Should revert after `expiryTimestamp`.
  ☑ Negative test?

Preconditions

- Assumes `option` exists.
- Assumes `option` has not yet expired.
- `msg.sender` should receive a `claim` token for this particular `option`.
- Assumes `safeTransferFrom` reverts on failure.
- Assumes the way the buckets are organized when writing to them is fair and that there's no way to manipulate the order of the buckets. Currently, there might be an issue where smaller buckets are at an advantage, since they are more likely to be filled first. (So rather than supplying a lot of underlying to a single bucket, it might be better to supply less to multiple buckets.)

Inputs

- `claimId`:
  - **Control**: Full control.
  - **Authorization**: Should check that `decoded(claimId).optionId == optionID` is given as a parameter.
  - **Impact**: In case of nonzero, `msg.sender` should own this token.
- `amount`:
  - **Control**: Full control.
  - **Authorization**: To calculate the actual amount of `underlying` needed, they transfer from `msg.sender` the `(rxAmount + fee)` where `uint256 rxAmount = amount * optionRecord.underlyingAmount`; then the `msg.sender` is issued `amount` shares, which are eventually recalculated.
  - **Impact**: Refers to the `desired` amount of `options`, not the actual amount of underlying that is to be supplied.
- `optionId`:
  - **Control**: Full control - it's used as an `optionKey`.
  - **Authorization**: Checks that the option is valid and it has not expired.
  - **Impact**: N/A.

Function call analysis

- `SafeTransferLib.safeTransferFrom(ERC20(underlyingAsset), msg.sender, address(this), (rxAmount + fee));`
  - **What is controllable?** `underlyingAsset`, to some degree; `msg.sender` and `rxAmount` to some degree.
  - **If return value controllable, how is it used and how can it go wrong?** There isn't return value.
  - **What happens if it reverts, reenters, or does other unusual control flow?**

Writing of the options won't happen if this reverts, since no funds would have been transferred from `msg.sender`.

- `_mint(msg.sender, optionId, amount, "");` **CAN BE RE_ENTERED IN**
  - **What is controllable?** `optionId`, `amount`, `msg.sender`
  - **If return value controllable, how is it used and how can it go wrong?** There isn't return value.
  - **What happens if it reverts, reenters, or does other unusual control flow?** Cannot revert, unless totalSupply was reached (highly unlikely).
- `_batchMint(msg.sender, tokens, amounts, "");` **CAN BE RE_ENTERED IN**
  - **What is controllable?** Tokens, amounts - basically this is used when the `claimId` is 0, meaning the `claimNFT` also has to be supplied for the user.
  - **If return value controllable, how is it used and how can it go wrong?** There isn't return value.
  - **What happens if it reverts, reenters, or does other unusual control flow?** Can revert if the size of tokens and amounts are wrong or if `msg.sender` doesn't support ERC1155. Also does external call of `to` address, where `to` is `msg.sender` - potential reentrancy.
- `_addOrUpdateClaimIndex(optionTypeStates[optionKey], claimKey, bucketIndex , amount);`
  - **What is controllable?** `optionTypeStates[optionKey]` - controlled; `amount` - limited control; `bucketIndex`- limited control. Could be manipulated by exercising an option from a known bucket, such that a fresh bucket is created when writing afterwards.
  - **If return value controllable, how is it used and how can it go wrong?** There isn't return value.
  - **What happens if it reverts, reenters, or does other unusual control flow?** No problems.
- `_addOrUpdateBucket(optionTypeState, amount);`
  - **What is controllable?** `optionTypeState` - full control; `amount` - limited control.
  - **If return value controllable, how is it used and how can it go wrong?** In case of a mistake, it might return the wrong `bucketIndex`.
  - **What happens if it reverts, reenters, or does other unusual control flow?** No problems.
- `decodeTokenId(tokenId)`
  - **What is controllable?** `tokenId`, the parameter.
  - **If return value controllable, how is it used and how can it go wrong?**
- Used to determine the `optionKEY` and the `claimNum` attributed to a certain `token Id`.
  - Would be bad if two `tokenIds` point to the same thing.

- **What happens if it reverts, reenters, or does other unusual control flow?** No problems.

Function: `exercise()`

Intended behavior

- Exercises one's option. Swap the `underlyingAsset` to the `exerciseAsset`.
- Exercises the `amount` of `optionId`, transferring in the exercise asset from `msg.sender`, and transferring of the underlying asset to `msg.sender`.

Branches and code coverage

Intended branches:

- Balance of `optionId` tokens of `msg.sender` decrease by `amount` value.
  ☑ Test coverage
- Balance of `exerciseAsset` of contract increase by the `exerciseAmount * amount + fee` value.
  ☑ Test coverage
- Balance of `underlyingAsset` of `msg.sender` increase by the `underlyingAmount * amount` value.
  ☑ Test coverage
- Assure that `option` has not expired and that `option` can be exercised.
  ☑ Test coverage
- Transfer the `exerciseAsset` from `msg.sender` and `balanceOf[exerciseAsset][msg.sender]` should decrease.
  ☑ Test coverage
- Assure that `option` exists.
  ☑ Test coverage
- Should `burn` the amount of options the `msg.sender` wants to exercise.
  ☑ Test coverage

Negative behavior:

- If `expiryTimestamp` is equal to `block.timestamp`, the transaction will be reverted.
  ☑ Negative test?
- If `exerciseTimestamp` is in the future, the transaction will be reverted.
  ☑ Negative test?
- If `expiryTimestamp` is in the past, the transaction will be reverted.
  ☑ Negative test?
- If `optionId` doesn't exist, the transaction will be reverted.
  ☑ Negative test?

- If `msg.sender` doesn't have enough `exerciseAsset` tokens, the transaction will be reverted.
  ☑ Negative test?

Preconditions

- Assumes `option` exists and is exercisable.

Inputs

- `optionId`:
  - **Control**: Limited control, only an existing `optionId`.
  - **Authorization**: There is a check that the last 12 bytes are zero because otherwise, it is a `claimId`. Also, there are the checks that `expiryTimestamp` should be more than `block.timestamp` and `exerciseTimestamp` should be less than `block.timestamp` for the current `optionId`.
  - **Impact**: `msg.sender` should own the `optionId` tokens. Also `msg.sender` can control fields of this `optionId`.
- `amount`:
  - **Control**: Limited control.
  - **Authorization**: The `msg.sender` should have at least `amount` of the options.
  - **Impact**: Allows the caller to send the specified amount of `exerciseAsset` to contract and receive the specified amount of `underlyingAsset`.
- `msg.sender`:
  - **Control**: N/A/
  - **Authorization**: `optionId` tokens balanceOf `msg.sender` should be more or equal to the `amount` value.
  - **Impact**: Only owner of `optionId` should be able to call this function.

Function call analysis

- `SafeTransferLib.safeTransferFrom(ERC20(exerciseAsset), msg.sender, address(this), (rxAmount + fee));`
  - **What is controllable?** `exerciseAsset` to some degree - `msg.sender` and `(rxAmount + fee)` to some degree.
  - **If return value controllable, how is it used and how can it go wrong?** There isn't a return value.
  - **What happens if it reverts, reenters, or does other unusual control flow?** Function fails because `msg.sender` doesn't have enough `exerciseAsset` to back up the `exercise` request.
- `SafeTransferLib.safeTransfer(ERC20(underlyingAsset), msg.sender, txAmount`

);
  - **What is controllable?** `msg.sender`, `underlyingAsset`, and `txAmount`.
  - **If return value controllable, how is it used and how can it go wrong?** There isn't a return value.
  - **What happens if it reverts, reenters, or does other unusual control flow?** It's bad if this fails because it means there's not enough `underlying` to supply the `msg.sender` with, despite the fact that they're entitled to it.
- `_burn(msg.sender, optionID, amount);`
  - **What is controllable?** `optionId`, `amount`.
  - **If return value controllable, how is it used and how can it go wrong?** There isn't a return value.
  - **What happens if it reverts, reenters, or does other unusual control flow?** Will revert balance of `msg.sender` less than the `amount`.
- `_assignExercise(optionTypeState, optionRecord, amount)`
  - **What is controllable?** `optionTypeState` - indirect control; `optionRecord` - indirect control; `amount` to some degree.
  - **If return value controllable, how is it used and how can it go wrong?** There isn't a return value.
  - **What happens if it reverts, reenters, or does other unusual control flow?** Return if all buckets are empty.
- `decodeTokenId(optionId)`
  - **What is controllable?** `optionId`, the parameter.
  - **If return value controllable, how is it used and how can it go wrong?**
- Used to determine the `optionKEY` and the `claimKey` attributed to a certain `token Id`.
  - Would be bad if two `optionIds` point to the same thing.
- **What happens if it reverts, reenters, or does other unusual control flow?** No problems.

Function: `redeem()`

Intended behavior

- Allow `underlying` suppliers to redeem their claim for an option; `msg.sender` is to receive either
  1. the desired `exercise` amount
  2. mix of `exercise` amount and `underlying`
  3. full refund of `underlying`
- Shouldn't allow transferring arbitrary `underlying` (or collateral) unless this `underlying` was specifically `written` for this `optionID`! Otherwise, malicious `options`

could be leveraged.

Branches and code coverage

Intended branches:

- Balances of `underlying` assets of the `msg.sender` should be returned in full if their assets have not been exercised and balance of `exercise` hasn't changed.
  ☑ Test coverage
- Balances of `exercise` assets of the `msg.sender` should increase depending if all their `underlying` assets have been exercised.
  ☑ Test coverage
- Balances of `underlying` and `exercise` of the `msg.sender` should increase if their assets have been partially exercised.
  ☑ Test coverage
- Assure `claimRecord.amountWritten` is depleted.
  ☑ Test coverage
- Assure `optionID` exists.
  ☑ Test coverage
- `burn` the claim token of a user.
  ☑ Test coverage

Negative behavior:

- Shouldn't allow to redeem assets twice.
  ☑ Test coverage

Preconditions

- The `expiryTimestamp` has come.
- The `claimId` was minted over the `write` function.

Inputs

- `claimID`:
  - **Control**: Full control.
  - **Authorization**: Must assure that it exists and that the `option` attributed to it has not expired.
  - **Impact**: Allows the caller to redeem their claim for the `underlying` and `exercise` assets, so it is important to check that caller is the owner of `claimId` and that they didn't redeem it before.

Function call analysis

- `_getPositionsForClaim(optionKey, claimId, optionRecord)`
  - **What is controllable?** `claimId` and `optionKey`, because it is a part of `claimId`, but it is a limited control because the `msg.sender` should be the owner of `claimId`.
  - **If return value controllable, how is it used and how can it go wrong?** The return values are the number of tokens that will be sent to the `msg.sender`, so in case of wrong calculation, the caller could steal some tokens or receive less than expected.
  - **What happens if it reverts, reenters, or does other unusual control flow?** Will revert in case of wrong `claimBucketInfo.amountWritten` and `claimBucketInfo.amountExercised` calculations, when `amountExercised` will be more than `amountWritten`.
- `_burn(msg.sender, claimId, 1)`
  - **What is controllable?** `claimId`.
  - **If return value controllable, how is it used and how can it go wrong?** There isn't a return value.
  - **What happens if it reverts, reenters, or does other unusual control flow?** Will revert if `msg.sender` isn't owner of the `claimId` token.
- `SafeTransferLib.safeTransfer`
  - **What is controllable?** N/A.
  - **If return value controllable, how is it used and how can it go wrong?** There isn't return value.
  - **What happens if it reverts, reenters, or does other unusual control flow?** Will revert in case of error inside token contract.

Function: `sweepFees()`

Intended behavior

- Should allow the `feeTo` address to sweep the amassed fees.

Branches and code coverage

Intended branches:

- Should be callable by anyone; however, the `fees` should only be transferrable to `feeTo`.
  - ☑ Test coverage
- Decrease the balances of each token after they've been withdrawn.
  - ☐ Test coverage

Negative behavior:

- Shouldn't leave the `feeBalance` not updated.
  ☑ Negative test?

Preconditions

- Assumes some fees have accrued.
- Assumes there is actually enough `tokenBalance` to pay for the fees.

Inputs

- `tokens`:
  - **Control**: Full control - list of arbitrary addresses.
  - **Authorization**: There's a check that the internal balance (called `feeBalance[token]`) exists and has been increasing. Technically this should guard against draining tokens that shouldn't be drainable.
  - **Impact**: The address of the contract, which function will be called.

Function call analysis

- `SafeTransferLib.safeTransfer(ERC20(token), sendFeeTo, sweep);`
  - **What is controllable?** `ERC20(token)` and `sweep`.
  - **If return value controllable, how is it used and how can it go wrong?** N/A.
  - **What happens if it reverts, reenters, or does other unusual control flow?** Transfer of funds won't happen and entire transaction fails; won't be an issue since it can be easily reproduced.

Function: `_addOrUpdateBucket()`

Intended behavior

- Internal function. The function is called only from `write` function and returns the `bucketIndex` value.
- Gets `_claimBucketByOption` by `optionKey`. The `bucketIndex` is the current length of `_claimBucketByOption[optionKey]`.
- If `_claimBucketByOption[optionKey]` is empty, then add the new `OptionsDayBucket` object, update `_unexercisedBucketsByOption[optionKey]`, and return.
- Otherwise, it gets the last `OptionsDayBucket` from `_claimBucketByOption[optionKey]`.
- If the next day has come, add the new `OptionsDayBucket` object, update `_unexercisedBucketsByOption[optionKey]`, and return.

- The same actions as the case with empty `_claimBucketByOption[optionKey]`.
- If the same day, increase the current `amountWritten` by `amount`, check `_doesBucketIndexHaveUnexercisedOptions`, and update if it doesn't have one.

Branches and code coverage

Intended branches:

- Creates a `claimBucket` if one doesn't exist.
  ☑ Test coverage
- Covers all edge cases when querying for current `claimBuckets`.
  ☑ Test coverage
- Should be used when `writing` options; ensure that all fields are updated for the specific `optionKey`.
  ☑ Test coverage

Negative behavior:

- Shouldn't be callable after an `option` has expired. (This should be covered in `write`.)
  ☑ Negative test?

Preconditions

- Assumes it's only used when `writing` an option (basically when the claim is created).

Inputs

- `uint160 optionKey`:
  - **Control**: Full control.
  - **Authorization**: No checks at this level.
  - **Impact**: N/A.
- `uint112 amount`:
  - **Control**: Full control.
  - **Authorization**: No checks at this level.
  - **Impact**: N/A.

Function call analysis

- `_getDaysBucket()`
  - **What is controllable?** The `block.timestamp` to some degree.
  - **If return value controllable, how is it used and how can it go wrong?** No

problems.
  - **What happens if it reverts, reenters, or does other unusual control flow?** No problems.
- `**_updateUnexercisedBucketIndices( optionKey, bucketIndex, unexercised);**`
  - **What is controllable?** `optionKey` - used to `update` the unexercised bucket indices.
  - **If return value controllable, how is it used and how can it go wrong?** There isn't return value.
  - **What happens if it reverts, reenters, or does other unusual control flow?** No problems.

Function: `_addOrUpdateClaimIndex()`

Intended behavior

- Internal function. The function is called only from `write`.
  1. If `_claimIdToClaimIndexArray[claimId]` is empty, add new `OptionLotClaimIndex{amountWritten:amount, bucketIndex}` object and return.
  2. If `bucketIndex` of the last element is less than the passed `bucketIndex` value, then add a new `OptionLotClaimIndex{amountWritten:amount, bucketIndex}` object and return.
  3. Otherwise, increase the `amountWritten` value of the last element.

Preconditions

- Assumes it's used when `writing` an option.

Inputs

- `claimId`:
  - **Control**: Partial control.
  - **Authorization:** It's called within `write`; it's checked there.
  - **Impact**: All written tokens will be counted for this identifier.
- `bucketIndex`:
  - **Control**: Little control - used from the return of `addOrUpdateClaimBucket`.
  - **Authorization:** Should always exist; there's also a check on whether it's identical to last's index or not.
  - **Impact**: The index of the bucket in which the tokens were written.
- `amount`:
  - **Control**: Full control.

  - **Authorization**: It's basically the amount that's written, AKA the amount of options that the user buys.
  - **Impact**: The amount of written tokens.

Function call analysis

There aren't external calls.

Function: `_assignExercise()`

Intended behavior

- Performs fair exercise assignment via the pseudorandom bucket selection. If the exercise amounts more than the selected basket contains, the next bucket will be used.

Branches and code coverage

Intended branches:

- Completely exercised arrays shouldn't be accessible by index.
  ☑ Test coverage
- Should an `unexercisedBucket` be completely exercised, it should be removed.
  ☑ Test coverage
- Select an `index` and use that for assigning the `exercise`.
  ☑ Test coverage

Negative behavior:

- If no more picks can be made, it should `revert` without consuming the gas.
  ☑ Negative test?

Preconditions

- Assumes there are multiple buckets to choose from and all of them can be exercised.
- Assumes that the caller cannot pass more option than was written.

Inputs

- `OptionTypeState optionTypeState`:
- **Control**: Indirect control.
  - **Authorization**: No checks.
  - **Impact**: The value is used to get info about unexercised buckets.

- `Option optionRecord`:
- **Control**: Indirect control, from `optionTypeState`.
  - **Authorization**: No checks.
  - **Impact**: The value is used to get `settlementSeed`'s value to generate pseudorandom bucket index.
- `uint112 amount`:
  - **Control**: Partly controlled.
  - **Authorization**: All checks are inside the `exercise` function that calls this one. Caller cannot pass more than they own.
  - **Impact**: The amount of options that will be exercised.

Function call analysis

There aren't external calls.

Function: `_getAssetAmountsForClaimIndex()`

Intended behavior

- Should return the `exercised` and `unexercised` options for a given claim.
- Remove empty bucket from the `_unexercisedBucketsByOption[optionKey]` mapping.

Branches and code coverage

Intended branches:

- Return correct `underlyingAmount` and `exerciseAmount`.
  ☑ Test coverage

Preconditions

- Assumes that the option `claimIndex` belongs to a `claim` that can be exercised.

Function call analysis

There aren't external calls.

Function: `setFeeTo()`

Intended behavior

Allows current `FeeTo` address to change the `feeTo` address.

Branches and code coverage

Intended branches:

- After the call `newFeeTo` is set.
  ☑ Test coverage

Negative behavior:

- Revert if `newFeeTo` is zero address.
  ☑ Negative test?
- Revert if `msg.sender` is not `FeeTo`.
  ☑ Negative test?

Preconditions

- The current `FeeTo` is not zero address.

Inputs

- `newFeeTo`:
  - **Control**: Full control.
  - **Authorization**: Only current `FeeTo`.
  - **Impact**: In case of an error when changing the address, the fee will be lost.

Function call analysis

There aren't external calls.

5.2 File: `TokenURIGenerator`

Function: `generateNFT()`

Intended behavior

- Generate an `SVG` associated with an `NFT` option.

Branches and code coverage

Intended branches:

- Generates a unique `NFT` based on the given parameters (for the header, amounts and date sections).
  ☐ Test coverage

Negative behavior:

- Shouldn't return an unusable SVG. This could happen if there are `'` in the parameters names and so on.
  - ☐ Negative test?

Preconditions

- Assumes each generated SVG is unique.

Inputs

- `TokenURIParams params`:
  - **Control**: Full control - make sure that `params` is checked and not null; otherwise, stuff might break in the SVG. Also, there's no `'`.
  - **Authorization:** N/A.
  - **Impact**: Data to be added to the generated SVG for the NFT.

Function call analysis

- `ERC20(params.underlyingAsset).decimals();`
  - **What is controllable?** `params`.
  - **If return value controllable, how is it used and how can it go wrong?** Should be checked at protocol level.
  - **What happens if it reverts, reenters, or does other unusual control flow?** No problems.
- `ERC20(params.exerciseAsset).decimals();`
  - **What is controllable?** `params`.
  - **If return value controllable, how is it used and how can it go wrong?** Should be checked at protocol level.
  - **What happens if it reverts, reenters, or does other unusual control flow?** No problems.

Function: `constructTokenURI()`

Intended behavior

Generates a URI for a claim NFT.

Branches and code coverage

Intended branches:

- Ensures that URI is generated properly.
  - ☐ Test coverage

Negative behavior:

- Revert if `params.underlyingAsset` or `params.exerciseAsset`'s address is zero.
  - ☐ Negative test?

Inputs

- `TokenURIParams params`:
  - **Control**: Full control.
  - **Authorization:** N/A.
  - **Impact**: Data to be added to the generated URI.

Function call analysis

There aren't external calls here.

Function: `generateName()`

Intended behavior

Generates a name for the NFT.

Branches and code coverage

Intended branches:

- Ensures that the name generated properly.
  - ☐ Test coverage

Negative behavior:

- Reverts if `year`, `month`, or `day` is in the wrong format.
  - ☐ Negative test?

Inputs

- `TokenURIParams params`:
  - **Control**: Full control.
  - **Authorization:** N/A.
  - **Impact**: Data to be added to the generated name.

Function call analysis

There aren't external calls here.

Function: `generateDescription()`

Intended behavior

Generates a description for the NFT.

Branches and code coverage

Intended branches:

- Ensures that the description is generated properly.
  - ☐ Test coverage

Negative behavior:

- Reverts if `params.underlyingAsset` or `params.exerciseAsset`'s address is zero.
  - ☐ Negative test?

Inputs

- `TokenURIParams params`:
  - **Control**: Full control.
  - **Authorization:** N/A.
  - **Impact**: Data to be added to the generated description.

Function call analysis

There aren't external calls here.

6 Audit Results

At the time of our audit, the code was not deployed to mainnet EVM.

During our audit, we discovered three findings. Two findings were of medium impact and one finding was informational. Valorem Labs Inc acknowledged all findings and implemented fixes.

6.1 Disclaimers

This assessment does not provide any warranties about finding all possible issues within its scope; in other words, the evaluation results do not guarantee the absence of any subsequent issues. Zellic, of course, also cannot make guarantees about any additional code added to the assessed project after the audit version of our assessment. Furthermore, because a single assessment can never be considered comprehensive, we always recommend multiple independent assessments paired with a bug bounty program.

For each finding, Zellic provides a recommended solution. All code in these recommendations are intended to convey how an issue may be resolved (i.e., the idea), but they may not be tested or functional code.

Finally, the contents of this assessment report are for informational purposes only; do not construe any information in this report as legal, tax, investment, or financial advice. Nothing contained in this report constitutes a solicitation or endorsement of a project by Zellic.