Zellic

Valorem Options

Smart Contract Security Assessment

December 27th, 2022

Prepared for:

Alcibiades

Valorem Inc

Prepared by:

Katerina Belotskaia and Vlad Toie

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 Inc from November 30th to December 9th, 2022. During this engagement, Zellic reviewed Valorem Options' 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?
- Is there any way to trick the protocol into preferentially assigning options to a specific user?
- Are there any circumstances when an arbitrager could profit from a certain option?
- 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

Due to the time-boxed nature of security assessments in general, there are limitations in the coverage an assessment can provide.

- 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.

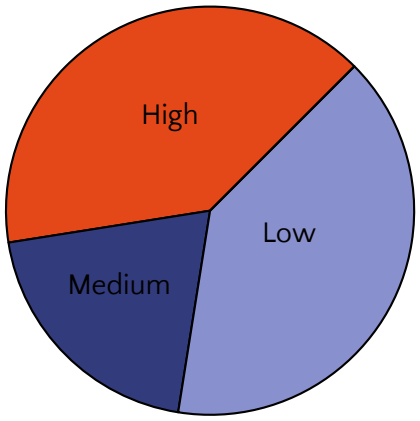
1.3 Results

During our assessment on the scoped Valorem Options contracts, we discovered four findings. An additional high impact finding was discovered by Valorem and verified by Zellic. No critical issues were found. Of the five findings, two were of high impact, one of medium impact, and the remaining findings were of low impact.

Additionally, Zellic recorded its notes and observations from the assessment for Valorem Inc's benefit in the Discussion section (4) and provided a full threat model description for various functions (5) at the end of the document.

Breakdown of Finding Impacts

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



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	`6c118f2090ba4f9ddac878a4afb1e5facb49b7ca`
Contracts	• OptionSettlementEngine • TokenURIGenerator
Type	Solidity
Platform	EVM

2.4 Project overview

Zellic was contracted to perform a security assessment with two consultants for a total of one-and-a-half person-weeks. The assessment was conducted over the course of two calendar weeks.

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:

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.

November 30, 2022	Kick-off call
November 30, 2022	Start of primary review period
December 9, 2022	End of primary review period

3 Detailed Findings

3.1 Un-encoded `claimID` can be used in `write()`

- **Target**: OptionSettlementEngine
- **Category**: Coding Mistakes
- **Likelihood**: Medium
- **Severity**: High
- **Impact**: High

Description

Users are allowed to create a position in an option through the `write()` function. This allows passing both the `optionID`, which corresponds to the option at hand, and the `claimID`, which corresponds to the claim that a user has when wanting to `redeem` the position.

Currently, the only performed check is that the lower 96 bytes of both the `claimID` and `optionID` are identical.

```
function write(uint256 optionId, uint112 amount, uint256 claimId)
    public returns (uint256) {
    (uint160 optionKey, uint96 decodedClaimNum)
    = decodeTokenId(optionId);

    // optionId must be zero in lower 96b for provided option Id
    if (decodedClaimNum ≠ 0) {
        revert InvalidOption(optionId);
    }

    // claim provided must match the option provided
    if (claimId ≠ 0 && ((claimId >> 96) ≠ (optionId >> 96))) {
        revert
    EncodedOptionIdInClaimIdDoesNotMatchProvidedOptionId(claimId,
    optionId);
    }
    // ...
```

If an attacker was to call `write()` with `claimID` identical to `optionID`, then they would effectively bypass the current checks, and instead of minting X options and one claim, they could mint `X + 1` options and no claim.

```
function write(uint256 optionId, uint112 amount, uint256 claimId)
    public returns (uint256) {
    uint256 encodedClaimId = claimId; // @audit-info assume the claimId
    has already been encoded.

    if (claimId == 0) {
        // ...
    } else { //
        // check ownership of claim

        uint256 balance = balanceOf[msg.sender][encodedClaimId];
        if (balance != 1) {
            revert CallerDoesNotOwnClaimId(encodedClaimId);
        }

        // retrieve claim
        OptionLotClaim storage existingClaim = _claim[encodedClaimId];

        existingClaim.amountWritten += amount;
    }
    // ...
    if (claimId == 0) {
        // Mint options and claim token to writer
        uint256[] memory tokens = new uint256[](2);
        tokens[0] = optionId;
        tokens[1] = encodedClaimId; // @audit-info assumes encodedClaimId
    is no longer the same as claimId
        // at this point, however, encodedClaimId = claimId = optionId

        uint256[] memory amounts = new uint256[](2);
        amounts[0] = amount;
        amounts[1] = 1; // claim NFT

        _batchMint(msg.sender, tokens, amounts, "");
```

Impact

Not minting the accompanying `claimNFT` leads to indefinitely locking the collateral that was associated with that particular claim.

Recommendations

We recommend assuring that `encodedClaimId` can never be the same as `optionID`.

Remediation

The issue has been fixed in commit 05f8f561.

3.2 Rounding error in the `redeem` mechanism

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

Description

During the redeem process, the `_getAmountExercised` function is called.

```
function _getAmountExercised(OptionLotClaimIndex storage claimIndex,
    OptionsDayBucket storage claimBucketInfo)
    internal
    view
    returns (uint256 _exercised, uint256 _unexercised)
{
    // The ratio of exercised to written options in the bucket multiplied
    by the
    // number of options actually written in the claim.
    _exercised = FixedPointMathLib.mulDivDown(
        claimBucketInfo.amountExercised,
        claimIndex.amountWritten,
        claimBucketInfo.amountWritten
    );

    // The ratio of unexercised to written options in the bucket
    multiplied by the
    // number of options actually written in the claim.
    _unexercised = FixedPointMathLib.mulDivDown(
        claimBucketInfo.amountWritten - claimBucketInfo.amountExercised,
        claimIndex.amountWritten,
        claimBucketInfo.amountWritten
    );
}
```

Due to the nature of how the amounts of exercised and unexercised options are calculated, there is the possibility of a rounding error. This may happen if `claimBucketInfo.amountWritten - claimBucketInfo.amountExercised * claimIndex.amountWritten < claimBucketInfo.amountWritten`. For example, this applies when the amount that was exercised globally has almost reached the amount that was written globally, and a users written claim is relatively low. In this case, the user will receive no underlying

tokens, even though they have exercised their options, as well as slightly less exercise tokens than they should have.

Impact

Depending on the variables of the equations, the user may potentially incur a loss of some or all of their `unexercised` or `exercised` tokens.

Recommendations

This issue was identified by the Valorem team and verified by Zellic. Valorem implemented changes to the calculations in `underlying()`, `redeem()`, and `claim()`, by placing all multiplication before division, to prevent loss of precision.

Remediation

During the remediation phase of the audit, Valorem implemented significant changes to the options' writing mechanism and overall contract architecture in order to address the issues that were identified. A thorough examination will be conducted during the next audit phase to confirm that these changes have effectively resolved the issue.

3.3 Writing during the exercise period may lead to arbitrage opportunities

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

Description

Currently, a user is allowed to `write` an option until its `expiry` date. The `exercise` period, however, lasts from the `exercise` to the `expiry` timestamps of an option.

```
function write(uint256 optionId, uint112 amount, uint256 claimId)
    public returns (uint256) {
    // ...
    Option storage optionRecord = _option[optionKey];

    uint40 expiry = optionRecord.expiryTimestamp;
    if (expiry == 0) {
        revert InvalidOption(optionKey);
    }
    if (expiry ≤ block.timestamp) {
        revert ExpiredOption(optionId, expiry);
    }
    // ...
}

function exercise(uint256 optionId, uint112 amount) external {
    // ...
    Option storage optionRecord = _option[optionKey];

    if (optionRecord.expiryTimestamp ≤ block.timestamp) {
        revert ExpiredOption(optionId, optionRecord.expiryTimestamp);
    }
    // Require that we have reached the exercise timestamp
    if (optionRecord.exerciseTimestamp ≥ block.timestamp) {
        revert ExerciseTooEarly(optionId,
    optionRecord.exerciseTimestamp);
    }
    // ...
}
```

This overlapping of the writing and exercising periods is prone to arbitrage opportunities. Due to the way the options' buckets are organized (per day), one can predict that should a specific `exercise` happen in today's bucket, writing to it leads to a guaranteed share of the `exercise` tokens.

Impact

The arbitrage opportunity does not lead to loss of funds for the user; however, it may lead to unexpected returns in terms of `exercise` tokens of an option.

Recommendations

We recommend either disallowing the writing of options during the exercise period or creating a time buffer such that only buckets that have been written at least one day prior to the current epoch can be exercised.

Remediation

During the remediation phase of the audit, Valorem implemented significant changes to the options' writing mechanism and overall contract architecture in order to address the issues that were identified. A thorough examination will be conducted during the next audit phase to confirm that these changes have effectively resolved the issue.

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

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

Description

The `_claim` mapping contains the `OptionLotClaimIndex` object for `claimId`. This points the `claimId` to a claim's indices in the `_claimIndexArray` array. The information is created during the `_addOrUpdateClaimIndex` call. During the `redeem` call, an internal `_getPositionsForClaim` is called, which in turn retrieves the exercise and underlying amounts of a claim.

```
function redeem(uint256 claimId) external {

    // ...

    (uint256 exerciseAmount, uint256 underlyingAmount)
    = _getPositionsForClaim(optionKey, claimId, optionRecord);

    // ...
}

function _getPositionsForClaim(uint160 optionKey, uint256 claimId,
    Option storage optionRecord)
    internal view returns (uint256 exerciseAmount,
    uint256 underlyingAmount) {
    OptionLotClaimIndex storage claimIndexArray
    = _claimIdToClaimIndexArray[claimId];

    for (uint256 i = 0; i < claimIndexArray.length; i++) {
        OptionLotClaimIndex storage = claimIndexArray[i];
        OptionsDayBucket storage claimBucketInfo
    = _claimBucketByOption[optionKey][claimIndex.bucketIndex];
        (uint256 amountExercised, uint256 amountUnexercised)
    = _getAmountExercised(claimIndex, claimBucketInfo);
        exerciseAmount += optionRecord.exerciseAmount * amountExercised;
        underlyingAmount += optionRecord.underlyingAmount
    * amountUnexercised;
```

```
    }
}
```

Impact

The `claimId` token is burned, but the storage still contains information about it. This information is no longer necessary, and in the expected behavior of the protocol, it will never be re-used.

Recommendations

To avoid further unexpected behavior, we recommend deleting the `_claimIdToClaimIndexArray[claimId]` object altogether.

```
function redeem(uint256 claimId) external {

    // ...
    (uint256 exerciseAmount, uint256 underlyingAmount)
    = _getPositionsForClaim(optionKey, claimId, optionRecord);
    delete _claimIdToClaimIndexArray[claimId];
    // ...
}
```

Remediation

During the remediation phase of the audit, Valorem implemented significant changes to the options' writing mechanism and overall contract architecture in order to address the issues that were identified. A thorough examination will be conducted during the next audit phase to confirm that these changes have effectively resolved the issue.

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

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

Description

The `_claim` mapping contains the `OptionLotClaim` object for `claimId`, namely, data on the number of written options and status of claim. This information is created during the `write` call. During the `redeem` call the `claimed` state is changed to `true`, but the `amountWritten` value is not reset to zero.

```
function redeem(uint256 claimId) external {
    (uint160 optionKey, uint96 claimNum) = decodeTokenId(claimId);

    if (claimNum == 0) {
        revert InvalidClaim(claimId);
    }

    uint256 balance = this.balanceOf(msg.sender, claimId);

    if (balance != 1) {
        revert CallerDoesNotOwnClaimId(claimId);
    }

    OptionLotClaim storage claimRecord = _claim[claimId];
    Option storage optionRecord = _option[optionKey];

    if (optionRecord.expiryTimestamp > block.timestamp) {
        revert ClaimTooSoon(claimId, optionRecord.expiryTimestamp);
    }

    (uint256 exerciseAmount, uint256 underlyingAmount)
    = _getPositionsForClaim(optionKey, claimId, optionRecord);

    claimRecord.claimed = true;

    // ...
```

Impact

The `claimId` token is burned, but the storage still contains information about it. This information is no longer necessary, and in the expected behavior of the protocol, it will never be used.

Recommendations

To avoid further unexpected behavior, we recommend deleting the `_claim[claimId]` object altogether.

```
function redeem(uint256 claimId) external {
    // ...
    claimRecord.claimed = true;
    delete _claim[claimId];
    // ...
}
```

Remediation

During the remediation phase of the audit, Valorem implemented significant changes to the options' writing mechanism and overall contract architecture in order to address the issues that were identified. A thorough examination will be conducted during the next audit phase to confirm that these changes have effectively resolved the issue.

4 Discussion

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

4.1 Reentrancy

Since there is a possibility of reentrancy during the `_mint` and `_batchMint` functions call, we recommend performing the `underlyingAsset` tokens transfer from `msg.sender` before the `mint` and `_batchMint` calls.

4.2 The `feeTo` address change may be erroneous

Currently, the `setFeeTo` function is done as

```
function setFeeTo(address newFeeTo) public {
        if (msg.sender ≠ feeTo) {
            revert AccessControlViolation(msg.sender, feeTo);
        }
        if (newFeeTo == address(0)) {
            revert InvalidFeeToAddress(newFeeTo);
        }
        feeTo = newFeeTo;
    }
```

We recommend to use a two-step address change to prevent changing the address to an erroneous one.

```
function setFeeTo(address newFeeTo) public {
        if (msg.sender ≠ feeTo) {
            revert AccessControlViolation(msg.sender, feeTo);
        }
        if (newFeeTo == address(0)) {
            revert InvalidFeeToAddress(newFeeTo);
        }
        _newFeeTo = newFeeTo;
    }
```

```
...

function acceptFeeTo() public {
    require(_newFeeTo == msg.sender);
    feeTo = msg.sender;
}
```

4.3 Calldata instead of memory for function arguments

The function `sweepFees` receive arguments in memory; however, the arguments in question are not altered. Arguments that are received as memory need to first be copied to memory, which adds extra gas cost. We recommend switching to calldata to reduce gas cost.

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: `underlying()`

Intended behavior

- Provide information about the position underlying a token, useful for determining the value.
- returns total amt of underlying and exercise assets currently associated with given `optionKey` for `optionId` or `claimId`.

Branches and code coverage

Intended branches:

- assumes that `decodeTokenId` call works fine
  ☑ Test coverage
- Ensure that the token it's searched for actually exists.
  ☑ Test coverage
- Ensure that the return values for not exercised `claimId` are expected (exercisePosition == 0; underlyingPosition == amount * underlyingAmount)
  ☑ Test coverage
- Ensure 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.

Inputs

- `tokenId`:
  - **Control**: full control
  - **Authorization**: checked whether it exists. also 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 `token Id`.
  - would be bad if two `tokenId` 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 a `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.
- `getPositionsForClaim(optionKey, tokenId, optionRecord)`
  - **What is controllable?** `optionKey, tokenId`
  - **If return value controllable, how is it used and how can it go wrong?** the return values depends on how much tokens was exercised from the bucket with `claimId` tokens and how much tokens was 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?** can be reverted due to underflow

Function: `encodeTokenId()`

Intended behavior

The pure function. Allow to calculate the `tokenId` value for given `optionKey` and `claimNum`

Branches and code coverage

Intended branches:

- Ensure that the return values 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

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

Branches and code coverage

Intended branches:

- Ensure that the return values 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 it doesn't already exist.

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 whether the `optionType` exists. Should revert if `true`.
  ☑ Test coverage. covered by `isOptionInitialized(optionKey)`

Negative behavior:

- Make sure `underlyingAsset` $\neq$ `exerciseAsset/`
  ☑ Negative test? Covered by `if-case`
- Revert if `expiryTimestamp` < 24h
  ☑ Negative test?
- Revert if exercise window is less then 24h
  ☑ Negative test?

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**: -
- `underlyingAmount`:
- **Control**: limited control
  - **Authorization**: underlyingToken.totalSupply() >= underlyingAmount
  - **Impact**: -
- `exerciseAsset`:
- **Control**: only existing contract address
  - **Authorization**: exerciseToken.totalSupply() >= exerciseAmount
  - **Impact**: -
- `exerciseAmount`:
- **Control**: limited control
  - **Authorization**: exerciseToken.totalSupply() >= exerciseAmount
  - **Impact**: -
- `expiryTimestamp`:
- **Control**: full control
  - **Authorization**: expiryTimestamp >= (block.timestamp + 1 days)
  - **Impact**: if `exerciseTimestamp` in the distant future, it will not be possible to call the `redeem` function!
- `exerciseTimestamp`:
  - **Control**: full control
  - **Authorization**: expiryTimestamp >= (exerciseTimestamp + 1 days)
  - **Impact**: it is possible to call the `write` and the `exercise` function 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`.
- Writes a specified amount of the specified option, returning claim NFT id.
- The following situation is possible if function `write` will be called with `optionId == claimId` and `balanceOf[msg.sender][optionId] == 1`, then the function won't revert, and a non-existent `OptionLotClaim storage existingClaim = _claim[encodedClaimId];` will be used, because there isn't check that `existingClaim` exists. after that, it will be impossible to call `redeem` for this `claimId`, and funds can be stuck.

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, he receives the expected amount of `optionId` and one claim NFT
  ☑ Test coverage
- Should be called successfully before `expiryTimestamp`
  ☑ Test coverage

Negative behavior:

- Assumes that a minted option(with specifically crafted `underlying`) cannot be maliciously used eventually to retrieve good `exercising` tokens somehow?
  - ☐ Negative test?
- Should revert if `msg.sender` doesn't have enough `underlyingAsset` tokens
  - ☑ Negative test?
- Should revert if the first 20 bytes of `claimId` are not the same as `optionId`
  - ☑ Negative test?
- Should revert if `msg.sender` doesn't own `claimId` token
  - ☑ Negative test?
- Should revert if `optionId` is invalid (the last 12 bytes is not zero)
  - ☑ Negative test?
- Should revert if `optionId` is not exist
  - ☑ Negative test?
- Should revert before `exerciseTimestamp`
  - ☑ 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.

Inputs

- `claimId`:
  - **Control**: full control
  - **Authorization**: should check that `decoded(claimId).optionId == optionID` given as parameter;
- basically there is a check for `((claimId >> 96) ≠ (optionId >> 96))` but if this can be somehow passed then it could be leveraged.
  - **Impact**: in case of non-zero, `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**: -

Function call analysis

- `SafeTransferLib.safeTransferFrom(ERC20(underlyingAsset), msg.sender, address(this), (rxAmount + fee));`
  - **What is controllable?** `underlyingAsset`, to some degree, `msg.sender` , `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(encodedClaimId, bucketIndex, amount)`
  - **What is controllable?** encodedClaimId - controlled, amount - limited control
  - **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

- `_addOrUpdateClaimBucket(optionKey, amount);`
  - **What is controllable?** optionKey - 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` eg. create for the next day if current is not past yet. but the index of bucket is not controllable.
  - **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 `tokenId`.
  - would be bad if two `tokenId` point to the same thing.
- **What happens if it reverts, reenters, or does other unusual control flow?** no problems

Function: `exercise()`

Intended behavior

- Exercise one's option. Swap the `underlying` 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` · `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:

- Shouldn't allow transferring arbitrary `underlying` UNLESS this `underlying` was SPECIFICALLY `written` for this `optionID`!; Otherwise, malicious `options` could be leveraged. this should be assured through `_assignExercise`
  - ☐ Negative test?
- Shouldn't allow burning `option` that doesn't exist.
  - ☐ Negative test?
- If `exerciseTimestamp` is equal `block.timestamp`, the transaction will reverted
  - ☐ Negative test?
- If `expiryTimestamp` is equal `block.timestamp`, the transaction will be reverted
  - ☐ Negative test?
- If `exerciseTimestamp` in the future, the transaction will be reverted
  - ☑ Negative test?
- If `expiryTimestamp` 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`.
  - **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**: -
  - **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` `(rxAmount + fee)` 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?** function fails bc `msg.sender` doesn't have enough `exerciseAsset` to back-up the `exercise` request.
- `SafeTransferLib.safeTransfer(ERC20(optionRecord.underlyingAsset), msg.sender, txAmount);`
  - **What is controllable?** `msg.sender`, `underlyingAsset`, `txAmount`
  - **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?** it's bad if this fails bc 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 return value
  - **What happens if it reverts, reenters, or does other unusual control flow?** will revert if msg.sender doesn't have enough
- `_assignExercise(optionKey, optionRecord, amount);`
  - **What is controllable?** `optionKey`, `amount` 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?** should return if all buckets are empty?

- `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 `tokenId`.
  - would be bad if two `tokenId` 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`

Branches and code coverage

Intended branches:

- Balances of `underlying` AND/OR `exercise` of the `msg.sender` should increase depending on the case! Make sure every case is covered.(3 cases)
  ☐ 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 transferring arbitrary `underlying` (OR COLLATERAL) UNLESS this `underlying` was SPECIFICALLY `written` for this `optionID`!; Otherwise, malicious `options` could be leveraged
  - this should be assured through `_getPositionsForClaim`.

  Negative test?

Preconditions

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

Inputs

- `claimID`:
  - **Control**: full
  - **Authorization**: must assure that it exists and that the `option` attributed to it has not expired.
  - **Impact**: Allows the caller to redeem his claim for the `underlying` and `exercise` assets, so it is important to check that caller is owner of `claimId` and that he 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 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` calculation, 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 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?** -
  - **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` un-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 contract which function will be called

Function call analysis

- `SafeTransferLib.safeTransfer(ERC20(token), sendFeeTo, sweep);`
  - **What is controllable?** `ERC20(token), 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: `_addOrUpdateClaimBucket()`

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.
- Else 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:

- Create a `claimBucket` if one doesn't exist.
  - ☑ Test coverage
- Covered 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:

- Assure that the info gets depleted when used.
  - ☐ Negative test?
- 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;
  - **Authorization**: no checks at this level
  - **Impact**:
- `uint112 amount`:
  - **Control**: full
  - **Authorization**: no checks at this level
  - **Impact**:

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: `_updateUnexercisedBucketIndices()`

Intended behavior

Internal function. The function is called only from `_addOrUpdateClaimBucket`.

The function allows adding `bucketIndex` to `_unexercisedBucketsByOption[optionKey]`, which maintains a mapping from option id to a list of the unexercised bucket (indices).

Also, the `bucketIndex` will be marked as an unexercised option inside `_doesBucketIndexHaveUnexercisedOptions[optionKey]`.

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.

Branches and code coverage

Intended branches:

Negative behavior:

- Assumes that the indices' `amountWritten` is eventually depleted when used.
    - ☐ Negative test?

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

- fair exercise assignment by selecting a claim bucket between initial creation of the option type and `today`. Buckets are then iterated from oldest to newest; the seed for pseudorandom index is updated accordingly on the option type.

Branches and code coverage

Intended branches:

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

Negative behavior:

- `uint112 amountAvailable = claimBucketInfo.amountWritten - claimBucketInfo.amountExercised;` Should never revert!
  - ☐ Negative test?
- Shouldn't be biased in any way.
  - ☐ Negative test?
- If no more picks can be made · it should `revert` without consuming the gas.
  - ☐ Negative test?

Preconditions

- Assumes there's multiple buckets to choose from and all of them can be exercised.
- Assumes that the functions calling this work fine
- Assumes that there's no way that two `unexercisedBucketIndices` can point to the same `claimBucketInfo`.

Inputs

- `uint160 optionKey`:
  - **Control**: controlled
  - **Authorization**: all checks are inside the `exercise` function that calls this one.
  - **Impact**: allows to get data about written buckets.
- `uint112 amount`:

- **Control**: controlled
- **Authorization**: all checks are inside the `exercise` function that calls this one.
- **Impact**: the amount of options which will be executed

Function call analysis

There aren't external calls.

Function: `_getAmountExercised()`

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 and up-to-date `claimBucket` and `claimIndex` information.
  - ☐ Test coverage

Negative behavior

- `claimIndex`
  - ☐ Negative test?
- `claimBucketInfo`
  - ☐ Negative test?

Preconditions

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

Function call analysis

There aren't external calls.

Function: `_getPositionsForClaim()`

Intended behavior

Allows to get the exercise and underlying amounts for a claim

Branches and code coverage

Intended branches:

- Ensure that the `exerciseAmount` and `underlyingAmount` calculated properly
  - ☐ Test coverage

Negative behavior:

- Shouldn't be callable in cases where the `claimId`, `optionKey` and `optionRecord` are not referring to the same `option`. This should be covered by the fact that `optionKey` is retrieved through `claimId`, and that `optionRecord` accesses the `option` at that `optionKey`.
  - ☐ Negative test?

Preconditions

- Assumes that `claimId`, `optionKey` and `optionRecord` are linked and not refer to the same `option` basically.

Function call analysis

There aren't external calls.

Function: `setFeeTo()`

Intended behavior

Allows to 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.

External call analysis

There aren't external calls.

5.2 File: `TokenURIGenerator`

Function: `generateNFT()`

Intended behavior

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

Branches and code coverage

Intended branches:

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

Negative behavior:

- Also why return string and then do bytes of it? maybe it could just return `bytes` and use that in `generateName`.
  - ☐ Negative test?
- 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` are checked and not null; otherwise stuff might break in the SVG also that 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:

- Ensure that uri generated properly
  - ☐ Test coverage

Negative behavior:

- revert if `params.underlyingAsset` or `params.exerciseAsset` 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:

- Ensure that the name generated properly
  - ☐ Test coverage

Negative behavior:

- revert 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:

- Ensure that the description generated properly
  - ☐ Test coverage

Negative behavior:

- revert if `params.underlyingAsset` or `params.exerciseAsset` 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 four findings. An additional high impact finding was discovered by Valorem and verified by Zellic. Of these, two were of high risk, one was of medium risk, and two were low risk. Valorem 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.