Zellic

Bond Protocol

Smart Contract Security Assessment

November 2nd, 2022

Prepared for:

Bond Labs

Prepared by:

Vlad Toie and Katerina Belotskaia

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 email us at hello@zellic.io or contact us on Telegram at https://t.me/zellic_io.

1 Executive Summary

Zellic conducted an audit for Bond Labs from October 26th to November 2nd, 2022.

Our general overview of the code is that it well-organized and structured. The code coverage was adequate, for the majority of the functions. The documentation was adequate, although it could be improved. The code was easy to comprehend, and in most cases, intuitive. There were some parts of the code, namely the SDA contracts, which were of higher complexity than most of the other contracts.

Zellic thoroughly reviewed the Bond Protocol codebase to find protocol-breaking bugs as defined by the documentation and to find any technical issues outlined in the Methodology section (2.2) of this document.

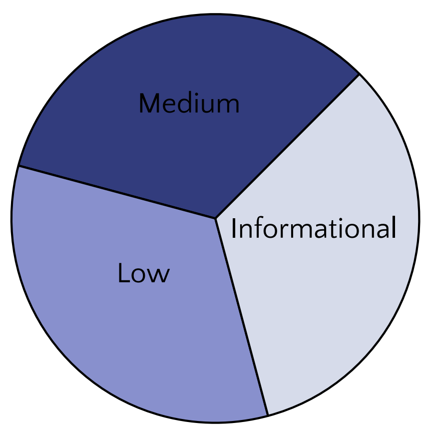
Specifically, taking into account Bond Protocol's threat model, we focused heavily on issues that would break core invariants such as the issuance and redemption of shares.

During our assessment on the scoped Bond Protocol contracts, we discovered 6 findings. Fortunately, no critical issues were found. Of the six findings, two were of medium severity, two were of low severity, and the remaining findings were informational in nature.

Additionally, Zellic recorded its notes and observations from the audit for Bond Labs's benefit in the Discussion section (4) at the end of the document.

Breakdown of Finding Impacts

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



2 Introduction

2.1 About Bond Protocol

Bond Protocol is a system to create Olympus-style bond markets for any token pair. The markets do not require maintenance and will manage bond prices based on activity. Bond issuers create BondMarkets that pay out a payout token in exchange for deposited quote tokens. Users can purchase future-dated payout tokens with quote tokens at the current market price and receive bond tokens to represent their position while their bond vests. Once the bond tokens vest, they can redeem it for the quote tokens.

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

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

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

Complex integration risks. Several high-profile 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. We perform a meticulous review of all of the contract's possible external interactions and summarize the associated risks: for example, flash loan attacks, oracle price manipulation, MEV/sandwich attacks, and so forth.

Code maturity. We review for possible improvements in the codebase 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 forth.

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

Similarly, Zellic organizes its reports such that the most important findings come first in the document rather than being 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 numerous soft factors, such as our clients' threat models, their business needs, their project timelines, and so forth. We aim to provide useful and actionable advice to our partners that consider their long-term goals rather than simply provide a list of security issues at present.

2.3 Scope

The engagement involved a review of the following targets:

Bond Protocol Contracts

Repository https://github.com/OlympusDAO/bonds

Versions `bd1626233463c5e776f1184d39981897d6ff12f0`

Programs

- BondAggregator
- ERC20BondToken
- BondFixedTermSDA
- BondFixedExpiryTeller
- BondFixedExpirySDA
- BondBaseTeller
- BondBaseCallback
- BondBaseSDA
- BondFixedTermTeller
- BondSampleCallback

Type Solidity

Platform EVM-compatible

2.4 Project Overview

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

Contact Information

The following project managers were associated with the engagement:

Jasraj Bedi, Co-founder
jazzy@zellic.io

Stephen Tong, Co-founder
stephen@zellic.io

The following consultants were engaged to conduct the assessment:

Vlad Toie, Engineer
vlad@zellic.io

Katerina Belotskaia, Engineer
kate@zellic.io

2.5 Project Timeline

The key dates of the engagement are detailed below.

October 26, 2022	Start of primary review period
November 2, 2022	End of primary review period

3 Detailed Findings

3.1 Lack of input validation

- **Target**: BondAggregator.sol
- **Category**: Business Logic
- **Likelihood**: Low
- **Severity**: Informational
- **Impact**: Low

Description

The `registerMarket` function does not validate `payoutToken_` and `quoteToken_` addresses for a zero address value.

Impact

Such a market will be impossible to use, so it is worth avoiding creating markets with zero token addresses.

Recommendations

We recommend implementing zero-address checks, such as the ones shown below:

```
function registerMarket(ERC20 payoutToken_, ERC20 quoteToken_)
        external
        override
        returns (uint256 marketId)
    {
        if (!_whitelist[msg.sender]) revert Aggregator_OnlyAuctioneer();
        require(payoutToken_ != address(0) && quoteToken_ != address(0),
    "zero address");
         ...
    }
```

Remediation

Bond Labs acknowledged this finding and implemented a fix in commit e9042fc7.

3.2 The bond `expiry_` can be in the past

- **Target**: BondFixedTermTeller, BondFixedExpiryTeller
- **Category**: Business Logic
- **Severity**: Medium
- **Likelihood**: Medium
- **Impact**: Medium

Description

There are two functions, namely `create()` and `deploy()`, available in both the `FixedTerm` and `FixedExpiry` tellers, which do not check whether the `expiry_` has passed the current `block.timestamp` or not.

In the case of the `deploy` function, this implies that a bond token can be created for a past `block.timestamp`, which could jeopardize the concept of bond tokens and their expiry.

```
function deploy(ERC20 underlying_, uint48 expiry_)
    external override nonReentrant returns (uint256) {
    uint256 tokenId = getTokenId(underlying_, expiry_);

    // @audit make sure that expiry_ is in the future.

    // Only creates token if it does not exist
    if (!tokenMetadata[tokenId].active) {
        _deploy(tokenId, underlying_, expiry_);
    }
    return tokenId;
}
```

For the `create` function, however, it implies that `bondTokens` would be issue for an already vested bond position.

Impact

In both of the aforementioned cases, having the `expiry_` in the past could potentially lead to bad user experience as well as undesired results in terms of bond issuance and redemption.

Recommendations

We recommend implementing checks that would block the issuance or deployment of `bondTokens` that have an expiry in the past.

```
require(expiry_ > block.timestamp, “error: expiry is in the past”);
```

Remediation

Bond Labs acknowledged this finding and implemented a fix in commits 4eb523da and 453d02e0.

3.3 Array indexes may be out of bounds

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

Description

In the `batchRedeem` function, two arrays are passed as parameters to the function. The two arrays, `tokenIds` and `amounts_`, are then accessed in one for loop for the same indices, without prior checking that their lengths are equal.

```
function batchRedeem(uint256[] calldata tokenIds_, uint256[]
    calldata amounts_) external override nonReentrant {
    uint256 len = tokenIds_.length;
    // @audit make sure that ther lengths are equal
    for (uint256 i; i < len; ++i) {
        _redeem(tokenIds_[i], amounts_[i]);
    }
}
```

Impact

Should there be a scenario when the lengths mismatch, the out-of-bounds error would trigger the function call to revert altogether at the last index, thus wasting the gas used for the transaction.

Recommendations

We recommend implementing a check such that the length of the arrays is properly checked before the for loop.

```
require(tokenIds.length == amounts_.length, “arrays' lengths mismatch”);
```

Remediation

Bond Labs acknowledged this finding and implemented a fix in commit 436d18ec.

3.4 Removal from `callbackAuthorized` is not conclusive

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

Description

The `callbackAuthorized` mapping dictates which `msg.sender` is allowed to perform `callbacks` on a specific market, and it is set via the `setCallbackAuthStatus` function. The status of this authorization is only checked when the market is created, despite the fact that the `msg.sender` can lose their rights to perform callbacks in the meanwhile, should the owner decide so.

Currently, there are no checks whatsoever, in any of the accompanying contracts, for whether the `msg.sender` is allowed to perform callbacks on a market.

```
function _createMarket(MarketParams memory params_)
    internal returns (uint256) {
    {
        // Check that the auctioneer is allowing new markets to be created
        if (!allowNewMarkets) revert Auctioneer_NewMarketsNotAllowed();

        // Ensure params are in bounds
        uint8 payoutTokenDecimals = params_.payoutToken.decimals();
        uint8 quoteTokenDecimals = params_.quoteToken.decimals();

        if (payoutTokenDecimals < 6 || payoutTokenDecimals > 18)
            revert Auctioneer_InvalidParams();
        if (quoteTokenDecimals < 6 || quoteTokenDecimals > 18)
            revert Auctioneer_InvalidParams();
        if (params_.scaleAdjustment < -24 || params_.scaleAdjustment > 24)
            revert Auctioneer_InvalidParams();

        // Restrict the use of a callback address unless allowed
        if (!callbackAuthorized[msg.sender] && params_.callbackAddr
    != address(0))
            revert Auctioneer_NotAuthorized();
    }
    // ...
}
```

Impact

Allowing previously whitelisted `msg.sender` to perform callbacks may result in undesired actions on behalf of the market it previously represented, potentially leading to financial losses.

Recommendations

We recommend assuring that once a user has been unwhitelisted, they can no longer perform actions on behalf of the market they originally represented.

Remediation

Bond Labs acknowledged this finding and implemented a fix in commit 00ddf327.

3.5 Data desynchronization

- **Target**: BondBaseCallback, BondBaseTeller
- **Category**: Business Logic
- **Severity**: Low
- **Likelihood**: Low
- **Impact**: Low

Description

When creating a market, the user can set the address of the `callback` contract that will process transfers of the owner's tokens. To do this, the user should be whitelisted, but deploying the `callback` contract is not under control by project contract. Therefore, it is not guaranteed that the user will specify the same address of `_aggregator` contract as the `BondBaseTeller` contract. As a result, there may be a desynchronization of the market data used to process the token transfer.

Impact

As a result of a user error, the market may be unusable since it is impossible to edit the corresponding market settings after creation.

Recommendations

For the expected operation of the `BondBaseCallback` contract independent of user actions, we recommend directly passing the `payoutToken` and `quoteToken` token addresses to the `callback` function.

Remediation

Bond Labs acknowledged this finding and implemented a fix in commit 252f64d8.

3.6 The `vesting` value for `BondFixedTerm` type of market can be incorectly set by the market creator

- **Target**: BondBaseSDA
- **Category**: Coding Mistakes
- **Likelihood**: Informational
- **Severity**: Informational
- **Impact**: Informational

Description

There are two types of markets for fixed term and fixed expiry bonds. The maximum `vesting` value for fixed term bonds is 50 years, but actually there are not any checks of the input `params_.vesting` value; therefore, the market creator can set greater value than the maximum.

Impact

A market created with a larger than expected `vesting` value is invalid, and it can cause unexpected behavior for this market. For example, the `isInstantSwap` function will define this market as an instant swap market and return `true` value.

Recommendations

Add validation of the input `params_.vesting` value.

Remediation

Bond Labs acknowledged this finding and implemented a fix in commit 0538adb3.

4 Discussion

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

4.1 Redundant call of `safeTransfer` for zero tokens value

The `claimFees` function will call the `token.safeTransfer(to_, send)` function even when the `send` value is zero. We recommend adding a check for zero value and in this case proceed to the next token address.

Remediation

Bond Labs acknowledged this finding and implemented a fix in commit 645e334b.

4.2 Assure token is active

In `redeem()` from `BondFixedTermTeller`, there is no explicit check on whether the token exists; however, the function call would still revert if the token did not exist. We recommend adding a check such that the validity of a token is explicitly checked.

Remediation

Bond Labs acknowledged this finding and implemented a fix in commit cdc23696.

4.3 Remove unchecked block

The unchecked block in the `purchaseBond()` function from `BondBaseSDA` comes after a check that would disallow any underflows or overflows happening:

```
// If amount/payout is greater than capacity remaining, revert
if (market.capacityInQuote ? amount_ > market.capacity : payout
    > market.capacity)
    revert Auctioneer_NotEnoughCapacity();

// @audit remove this unchecked;
unchecked {
    // Capacity is decreased by the deposited or paid amount
```

```
    market.capacity -= market.capacityInQuote ? amount_ : payout;

    // Markets keep track of how many quote tokens have been
    // purchased, and how many payout tokens have been sold
    market.purchased += amount_;
    market.sold += payout;
}
```

Despite that, we recommend removing the `unchecked` block for posterity reasons, in the case that further operations are performed around the `market.capacity`, which would, in turn, jeopardize the safety of this code block.

Remediation

Bond Labs acknowledged this finding and implemented a fix in commit ac5dbf77.

4.4 Double-counted auctioneer

The `registerAuctioneer` function allows to register a previously registered auctioneer. We recommend adding a check as shown below:

```
function registerAuctioneer(IBondAuctioneer auctioneer_)
    external requiresAuth {
        require(_whitelist[address(auctioneer_)] == false, "already
    registered");
        auctioneers.push(auctioneer_);
        _whitelist[address(auctioneer_)] = true;
    }
```

Remediation

Bond Labs acknowledged this finding and implemented a fix in commit 35687b13.

5 Threat Model

The purpose of this section is to provide a full threat model description for each function. We have analyzed every function in the smart contracts and created a written threat model. The threat model documents each function's externally controllable inputs and how an attacker could leverage each input.

5.1 BondAggregator.sol

Function `registerAuctioneer(IBondAuctioneer auctioneer_)`

1. **Intended behavior.**
   - Allows registering `auctioneers_` , only callable by Guardian.
   - Stores into an array(could maybe be duplicated, but it doesn't really matter)
   - Whitelists the auctioneer so that they can perform further actions (TBD)
   - only whitelisted callers can register new market
   - there isn't a possibility of removing the address from the whitelist
   - make sure the auctioneer hasn't been whitelisted before (!whitelisted[auctioneer])
2. **Negative behavior.**
   - Restriction on calls from untrusted users.

Function `registerMarket(ERC20 payoutToken_, ERC20 quoteToken_)`

1. **Intended behavior.**
   - Allows the whitelisted auctioneer contracts to register a new market.
   - Each market gets a unique `marketId`.
2. **Negative behavior.**
   - shouldn't allow to rewrite existing market - every new market will have new unique marketId, therefore, it is not possible to overwrite an existing one.
   - shouldn't allow to untrusted contract to register the market - there is a check that `msg.sender` should be whitelisted contract address, however the market can be created by anyone(eg. `BondFixedExpirySDA`) so what's the point of whitelisting in the first place? - markets can be created only using trusted SDA contacts.

- The market counter should increase whenever a new market has been created, so that no two markets can be overwritten.

3. **Preconditions.**
   - Assumes that `msg.sender` has been whitelisted beforehand (in this case that the aggregator calls)
   - Assumes that the `payoutToken` and `quoteTokens` are not 0 and are not equal (team said this is intended behavior)
4. **Postconditions.**
   - the new market should get a new unique `marketId` value
   - corresponding `IBondAuctioneer` address should be saved for `marketId` inside `marketsToAuctioneers` mapping
5. **Inputs.**
   - caller controls ERC20 payoutToken* and ERC20 quoteToken* addresses. Should check that addresses are not 0 and are not equal
6. **Examine all function calls the function makes.**
   - There are no function calls here.

5.2 BondFixedTermSDA.sol

This contract inherits the `BondBaseSDA` base contract.

Function `createMarket(bytes calldata params_)`

1. creates a new market via `BondBaseSDA`'s `_createMarket`.
2. apart from that, there seem to be no underlying functions that are inherited and can be called (since all of them would have to do with a theoretically created `Market` through its `id`)

5.3 BondFixedExpiryTeller.sol

Function `_handlePayout(address recipient, uint256 payout_, ERC20 underlying_, uint48) internal`

1. **Intended behavior.**
   - The function is supposed to handle the payout of `payout` tokens to the user. Not covered directly because it's an internal function.

2. **Negative behavior.**
   - It shouldn't transfer tokens to user if the vesting has not expired.
   - It shouldn't allow any `_underlying` to be transferred.
   - It shouldn't allow transfer on behalf of other users.
   - It shouldn't allow transfer of untrusted tokens.
   - It shouldn't call mint function of untrusted `bondToken`.
3. **Preconditions.**
   - That there exist enough `underlying_` tokens from the market creator/owner.
   - `bondToken` for corresponding `underlying_` token and `expiry` time should exist.
4. **Postconditions.**
   - user `bondTokens[underyling][expiry] balance ≥ payout_+ oldBalance;`

OR

- user `underlying balance ++` and `BondFixedExpiryTeller` `underlying balance--`

1. **Inputs.**
   - address recipient_: controlled - the destination of minted bondToken or transferred underlying.
   - uint256 payout_: controlled - the amount of `underlying` or minted bondTokens
   - ERC20 underlying_ : uncontrolled
   - uint48 vesting_: uncontrolled - period in time until which the bond mature
2. **Examine all function calls the function makes.**

   a. Call to `underyling.safeTransfer`:
   - What is controllable? (callee, params, return value): everything is controllable. However, it's an internal function, so all checks take place outside of this function
   - If the return value is controllable, how is it used, and how can it go wrong: There is no return value.
   - What happens if it reverts or tries to reenter: will be reverted if the contract doesn't have enough `underlying` tokens

   b. Call to `bondToken.mint`:
   - What is controllable? (callee, params, return value): recipient* is controllable, payout* is controllable.

- If return value controllable, how is it used and how can it go wrong: there is no return value.
- What happens if it reverts or tries to reenter: only in case `totalSupply` will reach the uint256 max value.

Function `create(ERC20 underlying_, uint48 expiry_, uint256 amount)`

1. **Intended behavior.**
   - This function is supposed to mint the bondToken pair of `underlying, expiry` to the sender, drawing `underlying` from the sender.
2. **Negative behavior.**
   - Disallow mint bond tokens with a `expiry` that's in the past. (**or** $\neq$ `0`)
   - Shouldn't leave the user with too little bondTokens minted (maybe add some slippage check)
   - Shouldn't allow infinite minting of the `bondToken`, since it could theoretically have the same `underlying` as other tokens.
3. **Preconditions.**
   - That a bond token already exists for the (`underlying, expiry_`) pair.
   - That the user has enough balance of `underlying_` to deposit.
4. **Postconditions.**
   - underlying_.balanceOf(address(BondFixedExpiryTeller)) · oldBalance+amount
   - bondToken.balanceOf(msg.sender) · oldBalance + (amount_ - fee)
   - if (protocolFee > createFeeDiscount) fee value should be assigned to the protocol owner
5. **Inputs.**
   - ERC20 underlying*: controllable - it's the underlying that's about to be sent to the contract (forming the `(underlying, expiry*)` pair. checked that a pair with it and the expiry exists
   - uint48 expiry_: controllable - it's part of the pair
   - uint256 amount_: controllable - it first needs to send the particular underlying from the user and then based on that it mints the bond tokens; apparently no need to check it
6. **Examine all function calls the function makes.**

   a. Call to `underlying_.balanceOf`
   - What is controllable? (callee, params, return value): address(this) - uncontrolled by the user, the return value is controllable.

- If return value controllable, how is it used and how can it go wrong: the user can transfer tokens directly to this address and increase the balance, but there is no bad impact.
- What happens if it reverts or tries to reenter: No problem.

b. Call to `underyling_.transferFrom(msg.sender, address(this), amount_)`

- What is controllable? (callee, params, return value): amount and underlying are controllable(internal function!)
- If return value controllable, how is it used and how can it go wrong: there is no return value.
- What happens if it reverts or tries to reenter: will be reverted if `msg.sender` doesn't have enough `underyling_` tokens.

c. Call to `bondToken.mint(msg.sender, amount_)`

- What is controllable? (callee, params, return value): `amount_` is controllable, but that's after the `transferFrom` happens, so it's safe.
- If the return value is controllable, how is it used, and how can it go wrong: there is no return value.
- What happens if it reverts or tries to reenter: No problem

Function `redeem(ERC20Bondtoken token_, uint256 amount_)`

1. **Intended behavior.**

   The function should enable users to redeem matured bond tokens issued by Bond Protocol for the vested underlying tokens. Acts like redeeming the rewards basically.

2. **Negative behavior.**

   - Don't redeem bonds which have not reached maturity
   - Don't redeem "counterfeit" bonds that aren't issued by Bond Protocol
   - Do not give out too many or too little underlying tokens
   - Don't give out the wrong underlying token
   - Don't give or take tokens from the wrong user

3. **Preconditions.**

   - The user has locked underlying tokens with the teller and received bond tokens in return.
   - The bonds have reached maturity.

4. **Postconditions.**

   - The user is now greater N underlying tokens.

- The user is now less N bond tokens.

5. **Inputs.**
   - token_: Full control (!). checked such that a `bondToken` exists for it.
   - amount_: Full control (!). no checks (!). However, invalid values will cause a revert (proven with unit test)
   - msg.sender: Can be any external sender, contract or EOA, no checks (!). Not an issue if msg.sender is external, but may be weird if msg.sender is the Teller itself
   - block.timestamp: Could be any time in the future (overapproximation). checked against token_.expiry()

6. **Examine all function calls the function makes.**

   a. Call to `token_.expiry()`
   - What is controllable: Fully controllable (callee is attacker supplied, can be any contract)
   - If return value controllable, how is it used and how can it go wrong: Bypasses block.timestamp maturity check.
   - What happens if it reverts or tries to reenter: No problem.

   b. Call to `token_.burn()`
   - What is controllable: Fully controllable (callee is attacker supplied, can be any contract)
   - If return value controllable, how is it used and how can it go wrong: N/A, return value is discarded.
   - What happens if it reverts or tries to reenter: No problem.

   c. Call to `token_.underlying()`
   - What is controllable: Fully controllable (callee is attacker supplied, can be any contract)
   - If return value controllable, how is it used and how can it go wrong: Attacker will control what underlying token is transferred.
   - What happens if it reverts or tries to reenter: No problem.

   d. Call to `token_.underlying().transfer()`
   - What is controllable: Fully controllable (callee is attacker supplied, can be any contract)
   - If return value controllable, how is it used and how can it go wrong: N/A, return value is discarded.
   - What happens if it reverts or tries to reenter: No problem.

Function `deploy(ERC20 underlying_, uinty48 expiry_)`

1. **Intended behavior.**
   - Allow users to create a new ERC20 token and associate it with `underlying` and `expiry_` values.
2. **Negative behavior.**
   - Disallow creating a pair of `underlying` and `expiry` that would overwrite already existing `bondToken`.
   - Disallow creating a pair with `expiry` in the past OR `expiry` different than `0` (they do have that condition in `handlePayout`)
   - Also there isn't any functionality to remove a `bondToken`
3. **Preconditions.**
   - That `underlying` exists (there's no check, however, that it's address 0)
   - That no bond for the `underlying, expiry` pair already exists
   - `decimals()` are standardized
   - That the `expiry_` is at least in the future
4. **Postconditions.**
   - That a `bondToken` is successfully created(if one did not previously exist)
   - That clone will be reverted in case of `bondToken` for these parameters already exists.
   - That the `bondToken[underlying_][expiry_]` stores the newly created contract address.
5. **Inputs.**
   - ERC20 underlying_: validation check that the pair of underlying, expiry doesn't already exist
   - uint48 expiry_: validation check that the pair of underlying, expiry doesn't already exist; also there is no check that is in the past.
6. **Examine all function calls the function makes.**

   a. Call to `underlying_.decimals()`
   - What is controllable? (callee, params, return value): there is no params value.
   - If return value controllable, how is it used and how can it go wrong: No problem.
   - What happens if it reverts or tries to reenter: deploy has nonReentrant modifier.

b. Call to `ERC20BondToken(address(bondTokenImplementation).clone(tokenData))`

- What is controllable? (callee, params, return value): tokenData is partly controllable.
- If return value controllable, how is it used and how can it go wrong: return the new bondToken address.
- What happens if it reverts or tries to reenter: should revert in case of token already exists.

5.4 BondFixedExpirySDA.sol

This contract inherits the `BondBaseSDA` base contract.

Function `createMarket(bytes calldata params_)`

1. this just creates a new market via `BondBaseSDA`'s `_createMarket`
2. apart from that, there seem to be no underlying functions that are inherited and can be called (since all of them would have to do with a theoretically created `Market` through its `id`)

5.5 BondBaseTeller.sol

Function `setReferrerFee(uint48 fee_) external`

1. Allow any user to set a fee value if their address will be used as `Referrer` address.
2. There is a limit on the maximum fee value.
3. This value can be used inside the `purchase` function to calculate `toReferrer` fee value. The user calling the `purchase` function controls the `referrer_` address theirself, so there are no problems associated with this.
4. Could be used to front-run users (as mentioned in audit #1)

Function `setProtocolFee(uint48 fee_) external requiresAuth`

1. **Intended behavior.**
   - This function allow authorized user or owner of contract to change protocolFee value.
   - There are no restrictions on the max amount of the fee

2. **Negative behavior.**
    - in case of a call by an unauthorized user or not the owner function should be rejected.
    - the contract owner can set 100% fee. even if this does not happen with the legitimate owner, in case of compromising the owner's key, the attacker can take all the funds sent to the contract by users since users also cannot reject transactions if they don't agree with the amount of the fee.
3. **Preconditions.**
    - if caller isn't owner of contract they must be approved for the call.
4. **Postconditions.**
    - protocolFee = fee_ changing the value to a new one.
5. **Inputs.**
    - uint48 fee_: controlled
6. **Examine all function calls the function makes.**

    a. `auth.canCall(msg.sender, address(this), msg.sig)`
    - What is controllable? (callee, params, return value): `auth` - the address of the contract specified in the constructor.
    - If return value controllable, how is it used and how can it go wrong?: if this function returns `true`, then the user is authorized to call this function.
    - What happens if it reverts or tries to reenter? will revert in case caller is unauthorized.

`claimFees(ERC20[] memory tokens_, address to_) external`

1. **Intended behavior.**
    - This feature allows any user to withdraw their rewards.
2. **Negative behavior.**
    - Shouldn't allow to withdraw someone else's tokens, only the assigned for caller amount of tokens.
    - Should reject if there is not enough token amount for withdraw.
    - Shouldn't call `safeTransfer` function if `send` value is zero. There isn't this kind of check here. should not reject the transaction but move on to the next token.
    - Shouldn't be able to transfer an arbitrary amount of tokens. This is properly checked since it queries for the `rewards[user]` when checking the amount.

3. **Preconditions.**
   - caller should have unspent reward for corresponding token address
   - Assumes that there isn't a way to double count the same reward token for a `user`
4. **Postconditions.**
   - The value of the assigned for caller amount of tokens should be reseted to zero. Also it should be done before `safeTransfer` call.
   - The caller receives no more tokens than they are supposed to.
   - The user's balance increased for each token by the amount they were supposed to receive.
   - The contract balance decreased for each token by the amount they previously sent to the user.
5. **Inputs.**
   - ERC20[] memory tokens_: controlled
   - address to_: controlled
6. **Examine all function calls the function makes.**

   a. Call to `token.safeTransfer(to_, send);`
   - What is controllable? (callee, params, return value): `token` - controlled, `to_` - controlled
   - If return value controllable, how is it used and how can it go wrong? there is no return value
   - What happens if it reverts or tries to reenter?: will be reverted if the contract doesn't have enough tokens

Function `purchase( address recipient_, address referrer_, uint256 id_, uint256 amount_, uint256 minAmountOut_ ) external nonReentrant`

1. **Intended behavior.**
   - allows users to exchange the `amount` of `quoteToken` to `payoutToken`.
   - calculates the fee values for `referrer_` if it exists and for `Protocol` if protocolFee was set.
   - calculate the `payout` value for `auctioneer` with passed `id_` for `amountLessFee` value.
   - saves fee values for later withdrawal
   - transfer from the `msg.sender` the `amount` of the `quoteToken` token to contract address.

- over `callbackAddr` address or directly transfer the `payout` value of the `payoutToken` from market `owner` to contract address.
- to `callbackAddr` address or directly transfer the `amountLessFee` value of the `quoteToken` from contract address to `owner` of market.
- transferring tokens directly to the `recipient_` address if it is `InstantSwap` or mint for the `recipient_` the `payout` amount of Bonds tokens which will be blocked for exchange to the `payoutToken` before the `expiry` time.

2. **Negative behavior.**
   - `msg.sender` sent an insufficient number of the `quoteToken` tokens
   - `payout` less then `minAmountOut_`
   - the market for the `id_` doesn't exist; assure that in `getMarketInfoForPurchase` from `BondBaseSDA`
   - the market owner doesn't have enough `payoutToken` tokens
   - if it isn't `InstantSwap` then `recipient_` shouldn't receive `payoutToken`, but only Bonds tokens
   - if it is `InstantSwap` then `recipient_` shouldn't receive the Bonds tokens, but only `payoutToken`.
   - Shouldn't allow transferring on behalf of other users.
3. **Preconditions.**
   - The market for the corresponding `id_` should be registered inside the `_aggregator` contract, to do this, the address of the `Auctioneer` which store the markets must be whitelisted inside `_aggregator` contract.
   - `msg.sender` should have enough `quoteToken` tokens · `amount`
   - market owner should have enough `payoutToken` · `payout`
4. **Postconditions.**
   - if `callbackAddr` $\neq$ `address(0)`, then `amountLessFee` value of `quoteToken` tokens should be successfully sent to `callbackAddr`
   - `msg.sender` balance of `quoteToken` decreased by `amount` quantity of tokens
   - `owner` balance of `payoutToken` decreased by `payout` quantity of tokens
   - if `InstantSwap` then `recipient_` should successfully receive the `payout` value of `payoutToken`
   - if it isn't `InstantSwap` then `recipient_` received the `payout` value of Bond tokens with the corresponding non-zero expiry time
   - `rewards` increased for non-zero `referrer_` by `toReferrer` amount of `quoteToken` tokens
   - `rewards` increased for `_protocol` by non-zero `toProtocol` amount of `quoteToken` tokens

5. **Inputs.**
   - address recipient_ - controlled. can be any address
   - address referrer_ - controlled. can be any address
   - uint256 id_ - controlled. used to select market
   - uint256 amount_ - controlled, amount of `quoteToken` which `msg.sender` should transfer to contract, there is a check inside the `_handleTransfers` function
   - uint256 minAmountOut* - controlled, min amount of `payout` tokens, which `recipient*`will receive, there is a check inside the`auctioneer.purchaseBond` function

6. **Examine all function calls the function makes.**

   a. Call to `IBondAuctioneer auctioneer = aggregator.getAuctioneer(id_)`
   - What is controllable? (callee, params, return value): `id_` - controlled, `msg.sender` can choose any created `auctioneer` by their `id_` value. only a whitelisted contract can become an `auctioneer`.
   - If return value controllable, how is it used and how can it go wrong? if `auctioneer` for current `id_` doesn't exist, then `getAuctioneer` function will return `address(0)` otherwise will return `auctioneer` address which has created the market.
   - What happens if it reverts or tries to reenter? `aggregator` - is a trusted contract address, no problems here

   b. Call to `auctioneer.getMarketInfoForPurchase(id_)`
   - What is controllable? (callee, params, return value): `id_` - controlled. it is unique market identification
   - If return value controllable, how is it used and how can it go wrong? address owner - market owner address can be any user. should transfer the necessary amount of `payoutToken`; address payoutToken - controlled, any user can create a market with any token address; address quoteToken - controlled, any user can create a market with any token address; uint48 vesting - controlled, any user can create a market with any vesting time. However, the values are controlled, the user must transfer the appropriate number of tokens for exchange. if these are dummy tokens, users will not exchange them.
   - What happens if it reverts or tries to reenter? No problem

   c. Call to `auctioneer.purchaseBond(id_, amountLessFee, minAmountOut_)`
   - What is controllable? (callee, params, return value): `id_` - controlled; `amountLessFee` - partially controlled because it's an amount value minus the fee values; `minAmountOut_` - controlled, if `payout` less than `minAmountOut_`, transaction should be rejected.

- If return value controllable, how is it used and how can it go wrong? as a result of calculation errors, the user may receive more `payout` tokens than he should; if `payout` value is less than `minAmountOut_`, the transaction should be canceled.
- What happens if it reverts or tries to reenter? Can revert in case if `payout` value is less than `minAmountOut_`, expected behavior.

d. Call to `handleTransfers(id, amount_, payout, toReferrer + toProtocol)`

- What is controllable? (callee, params, return value): `id_` - controlled; `amount_` - controlled, there is a check that `msg.sender` transfer no less than `amount_` to contract address; `payout` - uncontrolled, there is a check that owner transfer no less than `payout` to contract address; `toReferrer + toProtocol` - uncontrolled
- If return value controllable, how is it used and how can it go wrong?
- What happens if it reverts or tries to reenter?

f. Call to `handlePayout(recipient, payout, payoutToken, vesting)`: full review in the description of `BondFixedTermTeller.sol`

Function `_handleTransfers(uint256 id_, uint256 amount_, uint256 payout_, uint256 feePaid_) internal`

INTERNAL FUNCTION

1. **Intended behavior.**
   - Handles transfer of funds from user and market owner/callback
2. **Negative behavior.**
   - Shouldn't allow sending to an address different than market owner/ callback.
   - Shouldn't allow users to transfer CRAFTED tokens (via a malicious market for example) and retrieve useful tokens.(`as payout`). This could happen in `markets` from BondBaseSDA.
3. **Preconditions.**
   - `msg.sender` should approve to transfer `amount_` value of the `quoteToken` tokens to Teller contract.
   - That the quote tokens supplied by the `msg.sender` are perfectly fine, and they have been whitelisted/ accepted before, and that there is no way to supply dummy tokens in exchange for legitimate payout tokens.
   - `owner` of the market should approve transferring `payout` value of the `payoutToken` tokens to Teller contract.

4. **Postconditions.**
   - The `quoteToken.balanceOf[msg.sender]` should be depleted by `amount`, and the `quoteToken.balanceOf[callback OR owner of market]` should increase by amount after fees.
   - The `payoutToken.balanceOf[callback OR owner of market]` should be depleted by `payout_` and the `payoutToken.balanceOf[address(this)]` should increase by `payout_`
5. **Inputs.**
   - uint256 id_ - controlled
   - uint256 amount_ - controlled, if caller approved not enough tokens transaction will be rejected.
   - uint256 payout_ - uncontrolled, if the market owner approves not enough tokens transaction will be rejected.
   - uint256 feePaid_ - uncontrolled
6. **Examine all function calls the function makes.**

   a. Call to `aggregator.getAuctioneer(id).getMarketInfoForPurchase(id_);`
   - What is controllable? (callee, params, return value): `(address owner, address callbackAddr, ERC20 payoutToken, ERC20 quoteToken, , )` - it's not really controllable since it's supposedly whitelisted in the `getAuctioneer` function from the `aggregator`
   - If return value controllable, how is it used and how can it go wrong? uncontrolled
   - What happens if it reverts or tries to reenter? No problem

   b. Call to `quoteBalance = quoteToken.balanceOf(address(this))`
   - What is controllable? (callee, params, return value): uncontrolled
   - If return value controllable, how is it used and how can it go wrong? even if the caller controls the token and can manipulate with return value, this doesn't affect any users. In the case of using legitimate token address caller cannot manipulate this value.
   - What happens if it reverts or tries to reenter? No problem

   c. Call to `quoteToken.safeTransferFrom(msg.sender, address(this), amount_)`
   - What is controllable? (callee, params, return value): caller controls `amount_` value
   - If return value controllable, how is it used and how can it go wrong? There is no return value.
   - What happens if it reverts or tries to reenter? if the caller approved not enough tokens or the caller doesn't have enough tokens, then the transaction will be rejected

d. Call to `IBondCallback(callbackAddr).callback(id_, amountLessFee, payout_);`

- What is controllable? (callee, params, return value): it's supposed to handle the `payoutTokens` via the `callback` function back to the caller(**BondBaseTeller**); the `id_` and `payout_` params are directly controllable, being supplied through the `_handleTransfers` function.
- If return value controllable, how is it used and how can it go wrong? There is no return value.
- What happens if it reverts or tries to reenter? if this reverts, there are no `payoutTokens` transferred from the `callback`, and thus, the transaction itself fails.

5.6 BondBaseCallback.sol

`callback( uint256 id_, uint256 inputAmount_, uint256 outputAmount_ ) external`

1. **Intended behavior.**
   - Depending on the implementation of the `_callback` function, `callback` is supposed to send the `payoutTokens` back to the teller.
2. **Negative behavior.**
   - Shouldn't send the `payoutToken` to someone else other than the `teller`.
   - Shouldn't allow a teller with a different `aggregator` to call the function
3. **Preconditions.**
   - Assumes `msg.sender` is approved.
   - Assumes that a market for that `id` exists within the `aggregators'` markets
   - Assumes that the whitelist is the same as the aggregator's; OTHERWISE, it could theoretically be called by a malicious `msg.sender` since there are two different whitelists. That `msg.sender` could then exploit the contract and drain all `payoutToken`; they should use the Aggregator's whitelist just as they do with getting the markets
4. **Postconditions.**
   - `quoteToken.balanceOf(address(this)) += inputAmount, payoutToken_.balanceOf(address(this)) -= outputAmount`
   - `quoteToken.balanceOf(msg.sender) -= inputAmount, payoutToken_.balanceOf(msg.sender) += outputAmount`
   - `priorBalances` mapping should be updated properly (for both tokens)

5. **Inputs.**
    - uint256 id_ - controlled
    - uint256 inputAmount_ - controlled, but there is a check that the balance was increased by the corresponding value of the `quoteToken` tokens.
    - uint256 outputAmount* - controlled, there are no checks on the `outputAmount*`value, that is, any amount of`payoutToken`tokens can be sent to the`msg.sender`. Therefore, the market owner must be very careful with the whitelist of trusted callers.

6. **Examine all function calls the function makes.**
    - Call to `_callback(id, quoteToken, inputAmount_, payoutToken, outputAmount_)`: any logic implemented by the owner of the market (it's implementation agnostic); seems like the responsibility is shifted towards market owner.

5.7 BondBaseSDA.sol

Function `_createMarket(_createMarket(MarketParams memory params_) internal returns (uint256)`

1. **Intended behavior.**
    - Creates a new bond market.

2. **Negative behavior.**
    - add a check that `vesting` ≤ `MAX_FIXED_TERM` for `BondFixedTermTeller`
    - Shouldn't allow creating markets when `allowNewMarkets` is set to false.
    - When a market is created, its `id` must be unique (this is ensured in `aggregator`'s `createMarket` function)

3. **Preconditions.**
    - assumes markets can be created (`allowNewMarkets`)
    - assumes the MarketParams are properly checked
    - assumes `quoteTokens` and `payoutTokens` are not something sketchy (since anyone can create a market)
    - disallow registering a callback if a user is not whitelisted to do so. also, what if the status of a user changes? it's not reflected in the market the user owns, as they could still call the callback.

4. **Postconditions.**
    - The new market will be stored in a mapping `market [id]` → `MarketParams` object.

- The new market will have its own new market terms, market metadata and market params created.

5. **Inputs.**
   - The input is basically a set of `marketparams` that's used to store the info of a new market.

6. **Examine all function calls the function makes.**

   a. `uint256 marketId = aggregator.registerMarket(params.payoutToken, params_.quoteToken);`

   - What is controllable? (callee, params, return value): the parameters are controllable (supplied as function params to function where this is called); something to note is the fact that anyone can create a market for whatever `payoutToken` or `quoteToken` (even dummy ones); haven't yet found a way to exploit the protocol through this.
   - If return value controllable, how is it used and how can it go wrong: the return value is really important, so it must always increase (e.g. to not overwrite a previous market's details); this is ensured in the `registerMarket`. Hence it will always increase.
   - What happens if it reverts or tries to reenter: n/a

function `setIntervals(uint256 id_, uint32[3] calldata intervals_) external`

1. **Intended behavior.**
   - Set market intervals to different values than the defaults

2. **Negative behavior.**
   - shouldn't update metadata for an INEXISTENT ID (this is covered since `uint256(terms[id_].conclusion) - block.timestamp` would fail otherwise)

3. **Preconditions.**
   - assumes that a market exists for the `id` (including its metadata mapping)
   - shouldn't `tuneBelowCapacity` also be updated?
   - assumes that the owner calls the function(checked here `if (msg.sender ≠ market.owner) revert Auctioneer_OnlyMarketOwner();`)

4. **Postconditions.**
   - that the metadata for the market has been changed (IMPORTANT: not all fields are supposed to change)

5. **Inputs.**

- `uint256 id_` full control; despite that, `msg.sender` is checked to be the owner of the market
- `uint32[3] calldata intervals_`full control; limited checks(seems like all checks are fine)

6. **Examine all function calls the function makes.**
   - There are no function calls here.

function `purchaseBond(uint256 id_, uint256 amount_, uint256 minAmountOut_ ) external`

1. **Intended behavior.**
   - Exchange quote tokens for a bond in a specified market
2. **Negative behavior.**
   - Should only be callable by the `BondBaseTeller` inheriting contract, revert otherwise (this check is put in place)
   - Shouldn't allow the purchase of bonds from `ids` that have no registered market
   - Should disallow the purchase if the amount of bonds is less than the `minAmountOut` (this check is put in place) #slippagecheck
3. **Preconditions.**
   - Assumes that a teller is working properly and that the `msg.sender` is the teller in the cause.
   - Assumes that the market exists (through its `id`) and that it's not closed
   - Also, the check on `currentTime` should be > than `term.conclusion` since it may close within same block.
4. **Postconditions.**.
   - If `maxDebt` was reached, the market has to be closed, otherwise, `tuned` accordingly.
   - `market.capacity` will decrease based on the `amount` or `payout` values.
   - `market.purchase += amount_`
   - `market.sold += payout`
5. **Inputs.**
   - uint256 id_: full control, check that the market[id] has not concluded.
   - uint256 amount_: full control.
   - uint256 minAmountOut_: slippage check, is checked properly.

6. **Examine all function calls the function makes.**

   a. `(price, payout) = decayAndGetPrice(id, amount_, uint48(block.timestamp))`

   - What is controllable? (callee, params, return value): id - controllable; amount - controllable - affects the number of `payout` tokens; `block.timestamp` - partially controlled
   - If return value is controllable, how is it used and how can it go wrong: price - even if something goes wrong, the `price` cannot be less than the `market.minPrice`; payout - if it is calculated incorrectly and the capacity value wasn't set (in a case when it was set for quoteToken) and `market.minPrice` value wasn't set properly by the market owner, then it can be possible to steal the owner's funds.
   - What happens if it reverts or tries to reenter: not a problem.

function `decayAndGetPrice( uint256 id_, uint256 amount_, uint48 time_ ) internal`

1. **Intended behavior.**
   - Adjust the SDA and the pricing of a specific market id.
2. **Negative behavior.**
   - It shouldn't be callable whenever
   - It shouldn't be able to update inactive/ inexistent markets
   - The market capacity shouldn't overflow/ underflow from the `unchecked` scope.
3. **Preconditions.**
   - Assumes the market exists and is functional
   - Assumes adjustments can be made.
4. **Postconditions.**
   - The `totalDebt` should be updated (is updated twice)
   - The `lastDecay` should be updated
   - The `metadata[id_].lastDecay` should be updated
5. **Inputs.**
   - id - controllable; amount - controllable: affects the number of `payout` tokens; `block.timestamp` - partially controlled
6. **Examine all function calls the function makes.**
   - There are no external function calls here.

function `tune( uint256 id*, uint48 time_, uint256 price_ ) internal`

1. **Intended behavior.**
   - Adjusting the control variable to hit market capacity
2. **Negative behavior.**
   - Shouldn't tune other markets than the one at `id`
   - Shouldn't omit any important variables that need to be updated when `tuned`
3. **Preconditions.**
   - Assumes that the market can be tuned(which means that there's still debt to be purchased); basically if maxdebt hasn't been reached
4. **Postconditions.**
   - Update `markets[id_].maxPayout`, `terms[id_].controlVariable`, `metadata[id_].lastTune` , `metadata[id_].tuneBelowCapacity`, `metadata[id_].lastTuneDebt`
5. **Inputs.**
   - uint256 id_: full control.
   - uint256 time_: partially controlled.
   - uint256 price_: uncontrolled.
6. **Examine all function calls the function makes.**
   - There are no function calls here.

5.8 BondFixedTermTeller.sol

Function `_handlePayout( address recipient, uint256 payout_, ERC20 payoutToken_, uint48 vesting_) internal`

INTERNAL FUNCTION

1. **Intended behavior.**
   - This function is supposed to handle the payout of `payoutToken_` to the user if vesting time is zero or to mint the corresponding number of `Bond` tokens with the possibility of withdrawal after a certain amount of time.
   - It should transfer `payoutToken_` to `recipient_` directly if there is no vesting time
   - It should mint `Bond` tokens to `recipient_` if vesting time isn't zero

- If there isn't `Bond` token for current `payoutToken_` and `expiry` time, it should be created before mint

2. **Negative behavior.**
   - It shouldn't mint and transfer zero value of tokens.
   - It shouldn't mint and transfer to zero address.
   - It shouldn't transfer tokens if there are not enough of them on the contract balance. (The `safeTransfer` call will be reverted)
   - it's not supposed to mint if vesting is `==0` or transfer payout if vesting is $\neq$`0`.

3. **Preconditions.**
   - There is enough `payoutToken_` for transfer to `recipient`
   - Assumes that the token that is supposed to be minted exists. Otherwise, it tries creating it.
   - Assumes that `getTokenId` works well and there are no issues with how it retrieves the `id`.
   - Assumes that the `tokenId` is unique and no multiple `payoutToken_, expiry` pairs can exist.

4. **Postconditions.**
   - if vesting time is zero `payoutToken_.balanceOf(recipient_) ≤ balanceBefore+payout` and `payoutToken_.balanceOf(BondFixedTermTeller) ≥ balanceBefore-payout` and `recipient_` shouldn't get additional `Bond` tokens
   - if vesting time isn't zero `BondFixedTermTeller.balanceOf(recipient) ≤ balanceBefore+payout` and shouldn't get additional `payoutTokens`
   - if the `bondToken` wasn't deployed beforehand, now it should be.

5. **Inputs.**

   Since the function is internal, conclusions were made based on the analysis of the calling function.
   - address recipient_ - controlled
   - uint256 payout_ - it is partially controlled because it is calculated based on the controlled `amount` value. but this function should not care whether this value is calculated correctly.
   - ERC20 payoutToken_ - partially controlled, the caller selects any market to which the payoutToken address is linked, that is, the caller can select any address from those already linked to the markets.
   - uint48 vesting_ - it is not controlled because it is a market setting.

6. **Examine all function calls the function makes.**

a. Call to `_mintToken(recipient, tokenId, payout_);` calls `mint(to, tokenId_, amount_, bytes(“”))` calls `ERC1155TokenReceiver(to).onERC1155Received` If the calling contract is a contract, then it will be called.

- What is controllable? (callee, params, return value): `recipient` - controlled, and if `recipient` is the contract address, it will be called by mint hook ˆ see above; `tokenId` - partially controlled, calculated based on `payoutToken_` address and `expiry` value; they assume it's unique and no multiple `payoutToken_, expiry` pairs can exist; `payout_` - partially controlled, calculated based on `amount_`
- If return value controllable, how is it used and how can it go wrong? There is no return value.
- What happens if it reverts or tries to reenter? this function is called only at the end of the `purchase` function, and all important functions include `purchase` have `nonReentrant` protection. it looks like it can't be used, but it's better to keep this possibility in mind; it will be reverted in the following situations; if address `recipient == address(0)`; if `ERC1155TokenReceiver(to).onERC1155Received` will return wrong selector or reject the call inside

b. Call to `uint256 tokenId = getTokenId(payoutToken_, expiry)`

- What is controllable? (callee, params, return value): ERC20 payoutToken_ - partially controlled, the caller selects any market to which the payoutToken address is linked, that is, the caller can select any address from those already linked to the markets.; expiry - it is not controlled
- If return value controllable, how is it used and how can it go wrong? if there were possible collisions between real and fake tokens, then it would be possible to deposit dummy tokens and withdraw real ones; if there is any way to bypass the encoding and calculation, it would be possible to create an additional `payoutToken, expiry_` pair that would have the same `tokenId`
- What happens if it reverts or tries to reenter? No problems

c. Call to `_deploy(tokenId, payoutToken_, expiry)`

- What is controllable? (callee, params, return value): uint256 tokenId - it is not controlled; ERC20 payoutToken_ - partially controlled, the caller selects any market to which the payoutToken address is linked, that is, the caller can select any address from those already linked to the markets; expiry - it is not controlled
- If return value controllable, how is it used and how can it go wrong? there is no return value
- What happens if it reverts or tries to reenter? No problem

d. Call to `payoutToken_.safeTransfer(recipient_, payout_)`

- What is controllable? (callee, params, return value): payoutToken* - partially controlled, the caller can select any address from those already linked to the markets; payout* - controlled
- If return value controllable, how is it used and how can it go wrong? there is no return value
- What happens if it reverts or tries to reenter? will be reverted if the current contract doesn't have enough tokens

Function `create(ERC20 underlying_, uint48 expiry_, uint256 amount_)`

NO UNIT-TEST or at least not direct test

1. **Intended behavior.**
   - This function allows any user to get the `amount` value of the `Bond` tokens in exchange for `underlying_` ERC1155 tokens.
2. **Negative behavior.**
   - Shouldn't `mint` `amount` of tokens if a smaller quantity of `underlying_` tokens was deposited.
   - Shouldn't `mint` the `tokenId` tokens if `tokenMetadata[tokenId]` doesn't exist because otherwise, it would be possible to deposit dummy tokens and then create metadata with a real token and withdraw it. Also, it means that no one should be able to edit already created items of `tokenMetadata[tokenId]`.
   - Should be rejected if the caller doesn't have enough `underlying_` tokens.
   - Should be rejected if the caller doesn't get the expected value of `Bond` tokens. there is no such check here, but since the fee percentage is unlimited and can be changed at any time, the caller may not agree with the withdrawn fee value. we could add a slippage check here, just as in the other BondFixedExpiryTeller.
   - Shouldn't emit expired `bondTokens` since that means that they've vested already.
3. **Preconditions.**
   - `tokenMetadata[tokenId]` should be already created for current `underlying_` and `expiry_` values.
   - the caller should have enough `underlying_` tokens to pay for a deposit.
   - the correct `tokenId` should be minted (calculated based on the pair between the `payoutToken, expiry`.
   - should make sure that `expiry` is not in the past? as in, the bondToken pair had already vested?

4. **Postconditions.**
   - `underlying_.balanceOf(BondFixedTermTeller)` ≥ oldBalance+amount
   - `BondFixedTermTeller.balanceOf(msg.sender) ≤ balanceBefore+payout`
5. **Inputs.**
   - ERC20 underlying_ : controlled (!), it can be any address of ERC20 token, there is should exist `tokenMetadata` for this token address and `expiry` value, but it is possible to do over `deploy` function.
   - uint48 expiry : controlled (!)
   - uint256 amount_ : controlled (!)
6. **Examine all function calls the function makes.**

   a. Call to `uint256 tokenId = getTokenId(underlying_, expiry_);`
   - What is controllable? (callee, params, return value): `underlying_, expiry_` are controllable
   - If return value controllable, how is it used and how can it go wrong? `tokenId` should be unique for corresponding `underlying_, expiry_` values
   - What happens if it reverts or tries to reenter? No problem

   b. Call to `_mintToken(msg.sender, tokenId, amount_)` see `_handlePayout._mintToken` description

Function `redeem(uint256 tokenId, uint256 amount_) public basically(it does a function call to internal fct)`

1. **Intended behavior.**
   - The function should enable users to redeem matured bond tokens issued by Bond Protocol for the vested underlying tokens.
   - The function should burn the corresponding amount of `tokenId` tokens.
   - And transfer to `msg.sender` the same amount of `payoutToken`.
2. **Negative behavior.**
   - Don't redeem bonds that have not reached maturity: `if (block.timestamp < meta.expiry) revert Teller_TokenNotMatured(meta.expiry);`
   - Don't redeem "counterfeit" bonds that aren't issued by Bond Protocol: `_burnToken` called only for local bonds tokens
   - Do not give out too many or too few underlying tokens: it is possible to send only the amount of `payoutToken` that is available on the `Bond` balance of `msg.sender` `balanceOf[msg.sender][tokenId] -= amount;` otherwise, the transaction will be rejected on this line

- Don't give out the wrong payoutToken token: `payoutToken` address taken from `tokenMetadata` for corresponding `tokenId`. An attacker can add any address of `payoutToken` to the `tokenMetadata`, but because of the `_burnToken` function call, they can only redeem their tokens.
- Don't give or take tokens from the wrong user.

3. **Preconditions.**
   - The user has locked `payoutToken` tokens with the teller and received bond tokens in return with `tokenId` which connected with this `payoutToken` and `expiry` value.
   - The bonds have reached maturity.
   - Assumes that the `tokenid` has active metadata (it would fail otherwise anyway due to the `burnToken` function, there'd be an underflow there)
   - The bonds could have not been infinitely created; since the multiple `bondTokens` can be created for the same `payoutToken` this means that there might be a way to drain the contract if there is a way to craft infinitely many `bondTokens`!
4. **Postconditions.**
   - The user has now more underlying tokens.
   - The user has now less bond tokens.
   - the protocol should still have some `underlying` tokens left to pay the other users.
5. **Inputs.**
   - tokenId_: controlled,
   - amount_: controlled,
6. **Examine all function calls the function makes.**

   a. Call to `burnToken(msg.sender, tokenId*, amount_);`
   - What is controllable? (callee, params, return value): `msg.sender`; `tokenId` - directly controlled; COULD BE USER TO burn arbitrary bond tokens, however, they would have to be minted via `create` in the first place; `amount_` - controlled
   - If return value controllable, how is it used and how can it go wrong? there is no return value
   - What happens if it reverts or tries to reenter? No problem

   b. Call to `meta.payoutToken.safeTransfer(msg.sender, amount_)`
   - What is controllable? (callee, params, return value): `meta.payoutToken` - controlled; could be used to drain arbitrary `underlying` tokens, however,

the bondTokens issued for them would have to be burned in the first place, so no profit could really be made; `msg.sender`; `amount_` - controlled
- If return value controllable, how is it used and how can it go wrong? there is no return value
- What happens if it reverts or tries to reenter? No problem

Function `batchRedeem(uint256[] calldata tokenIds_, uint256[] calldata amounts_) external`

NO UNIT-TEST

- Essentially performs multiple `redeem` calls, which could have been done manually. A good check to mention here would be that `tokenIds` array length has to be equal to `amounts` array length.

Function `deploy(ERC20 underlying_, uint48 expiry_) external`

NO UNIT-TEST or at least not direct test

1. **Intended behavior.**
   - "Deploy" a new ERC1155 bond token for an (underlying, expiry) pair and return its address.
2. **Negative behavior.**
   - It shouldn't allow modifying an already existing `underlying, expiry` pair!
   - It must assure that the `expiry` is in the future!
3. **Preconditions.**
   - Assumes that the `underlying, expiry` pair doesn't already exist as a bondToken
   - That `getTokenId` calculates the `tokenId` properly and there's no way to bypass and create an additional token with same `id`.
4. **Postconditions.**
   - Assumes a new ERC1155 tokenId has been created with the `underlying, expiry` pair.
5. **Inputs.**
   - ERC20 underlying_: controlled,
   - uint48 expiry_: controlled,
6. **Examine all function calls the function makes.**
   - There are no function calls here.

Function `mintToken(address to,uint256 tokenId_,uint256 amount_) interna l`

INTERNAL FUNCTION

1. **Intended behavior.**
   - The internal function, which allows minting `amount_` of Bond `tokenId_` tokens for `to` address and increase `supply` value for this `tokenId_`.
2. **Negative behavior.**
   - Users should not have unrestricted access to this function. the calling function must control the all parameters, not allowing unlimited minting of any tokens.
3. **Preconditions.**
   - User should send the corresponding amount of `underlying` tokens to contract before the `mint`
4. **Postconditions.**
   - the user's `bondTokens` balance should increase by `amount`
   - the issued token's supply within the `metadata` mapping should increase by the value that was minted `tokenMetadata[tokenId_].supply += amount_;`
5. **Inputs.**
   - user should have possibility to mint `amount_` of tokens only in exchange for a corresponding `amount_` of `underlying` tokens
   - `tokenId_` value should be connected with `underlying` token address. Users shouldn't be able to deposit one type of `underlying` token and get other Bond tokens in return.
6. **Examine all function calls the function makes.**

   a. Call to `mint(to, tokenId_, amount_, bytes(""))` and `ERC1155TokenReceiver(to).onERC1155Received()` see `_handlePayout._mintToken` description

Function `_burnToken(address from,uint256 tokenId_,uint256 amount_) inte rnal`

INTERNAL FUNCTION

1. **Intended behavior.**
   - The internal function, which allows burning `amount_` of Bond `tokenId_` tokens from `from` address and decreases `supply` value for this `tokenId_`.

2. **Negative behavior.**
   - users should not have unrestricted access to this function. The caller function should control from parameter, not allowing users to burn other user's tokens.
   - shouldn't allow burning tokens if the user does not have enough tokens, partial burn should be impossible. should reject in this case

3. **Preconditions.**
   - user `from`should have · `amount` of `tokenId_` tokens.
   - the caller function should not allow to user pass any `from` address.

4. **Postconditions.**
   - the supply pf the `bondToken` should decrease by `amount`
   - the balance of the user in terms of `bondTokens` should decrease by `amount`.

5. **Inputs.**

6. **Examine all function calls the function makes.**

   a. Call to `burn(from, tokenId_, amount_)`
   - What is controllable? (callee, params, return value): the calling function should not allow the user to control the `from` value
   - If return value controllable, how is it used and how can it go wrong? there is no return value
   - What happens if it reverts or tries to reenter? N/A

Function `getTokenId(ERC20 underlying_, uint48 expiry_) public pure`

- Must return a unique identifier for the corresponding `underlying_` token address and `expiry_` value.

6 Audit Results

At the time of our audit, the code was deployed to mainnet Ethereum.

During our audit, we discovered six findings. Of these, two were of medium risk, two were low risk, and two were suggestions (informational). Bond Labs 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.