Zellic

Prepared for
Brank Dev
Avantis Labs, Inc.

Prepared by
Nipun Gupta
Kuilin Li
Zellic

February 23, 2024

Bloom Trading
Smart Contract Security Assessment

Contents

About Zellic

Zellic is a vulnerability research firm with deep expertise in blockchain security. We specialize in EVM, Move (Aptos and Sui), and Solana as well as Cairo, NEAR, and Cosmos. We review L1s and L2s, cross-chain protocols, wallets and applied cryptography, zero-knowledge circuits, web applications, and more.

Prior to Zellic, we founded the #1 CTF (competitive hacking) team ↗ worldwide in 2020, 2021, and 2023. Our engineers bring a rich set of skills and backgrounds, including cryptography, web security, mobile security, low-level exploitation, and finance. Our background in traditional information security and competitive hacking has enabled us to consistently discover hidden vulnerabilities and develop novel security research, earning us the reputation as the go-to security firm for teams whose rate of innovation outpaces the existing security landscape.

For more on Zellic's ongoing security research initiatives, check out our website zellic.io ↗ and follow @zellic_io ↗ on Twitter. If you are interested in partnering with Zellic, contact us at hello@zellic.io ↗.

1. Overview

1.1. Executive Summary

Zellic conducted a security assessment for Avantis Labs, Inc. from February 19th to February 22nd, 2024. During this engagement, Zellic reviewed Bloom Trading's code for security vulnerabilities, design issues, and general weaknesses in security posture.

1.2. 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:

- Did the new execution strategy introduce any vulnerabilities or security risks?
- Is the Blast yield distribution mechanism well-defined and implemented correctly?
- Are there any potential issues with the new funding rate calculations?
- Does the gas-claim logic align with expected behavior?

1.3. Non-goals and Limitations

We did not assess the following areas that were outside the scope of this engagement:

- Front-end components
- Infrastructure relating to the project
- Key custody

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

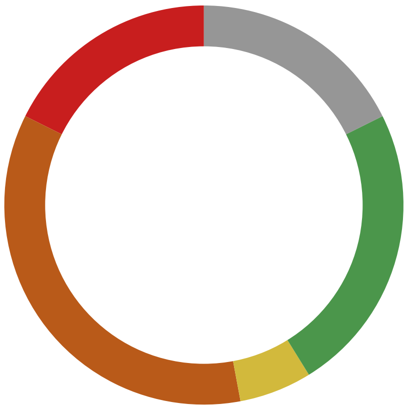
1.4. Results

During our assessment on the scoped Bloom Trading contracts, we discovered 17 findings. Three critical issues were found. Six were of high impact, one was of medium impact, four were of low impact, and the remaining findings were informational in nature.

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

Breakdown of Finding Impacts

Impact Level	Count
🟥 Critical	3
🟧 High	6
🟨 Medium	1
🟩 Low	4
⬜ Informational	3



2. Introduction

2.1. About Bloom Trading

Bloom Trading is developing a user-friendly leveraged trading platform, where users can long or short crypto using a financial primitive called perpetuals with native yields from Blast L2.

Synthetic leverage combined with a USDB stablecoin LP makes Bloom very capital efficient, allowing for high leverage (up to 100x). They are allowing LPs to leverage their capital by locking deposits in order to earn more fees.

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 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 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, upgradability 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.

Finally, Zellic provides a list of miscellaneous observations that do not have security impact or are not directly related to the scoped contracts itself. These observations — found in the Discussion (4. ↗) section of the document — may include suggestions for improving the codebase, or general recommendations, but do not necessarily convey that we suggest a code change.

2.3. Scope

The engagement involved a review of the following targets:

Bloom Trading Contracts

Repository	https://github.com/Bloom-Trading/bloom-contracts ↗
Version	bloom-contracts: `7999dbbb9754838958a9c15801a7272c35f0550f`
Programs	• Execute • PairInfos • PairStorage • PriceAggregator • Referral • Trading • TradingCallbacks • TradingStorage • Tranche • VaultManager • VeTranche
Type	Solidity
Platform	EVM-compatible

In addition to the above targets, we also reviewed a proposed upgrade to the Trading contract to add a Delegatable parent contract. The Delegatable source file version was `56c0cfff` ↗.

Furthermore, we have reviewed a modification concerning the transfer of fees to the treasury, specifying a 20% allocation of all fees. The relevant commits for these changes are `40e0955d` ↗ and `f9025875` ↗.

2.4. Project Overview

Zellic was contracted to perform a security assessment with two consultants for a total of eight person-days. The assessment was conducted over the course of four calendar days.

Contact Information

The following project manager was associated with the engagement:

Chad McDonald
Engagement Manager
chad@zellic.io ↗

The following consultants were engaged to conduct the assessment:

Nipun Gupta
Engineer
nipun@zellic.io ↗

Kuilin Li
Engineer
kuilin@zellic.io ↗

2.5. Project Timeline

The key dates of the engagement are detailed below.

February 19, 2024	Kick-off call
February 19, 2024	Start of primary review period
February 22, 2024	End of primary review period

3. Detailed Findings

3.1. Partial market close uses full amount for yield

Target	PairInfos		
Category	Coding Mistakes	**Severity**	Critical
Likelihood	Medium	**Impact**	Critical

Description

The function `getTradeValue` in PairInfos is called by TradingCallbacks in order to determine the value of a trade during closing as well as being used elsewhere:

```
function getTradeValue(
    ITradingStorage.Trade memory _trade,
    uint collateral,
    int percentProfit,
    uint closingFee
) external override onlyCallbacks returns (uint amount, int pnl, uint fees,
    uint lpYield) {
    // [...]

    uint accPendingYield = getPendingYield(_trade.initialPosToken,
    _trade.leverage, _trade.timestamp);

    (amount, pnl, fees, lpYield) = getTradeValuePure(
        collateral,
        percentProfit,
        r,
        accPendingYield,
        f,
        closingFee
    );

    emit FeesCharged( /* [...] */ );
}
```

When a market close is issued for an amount of collateral less than the collateral in the trade, the `collateral` parameter to this function is less than the `_trade.initialPosToken` — in all other cases, they are equal. However, the call to `getPendingYield` mistakenly uses the latter instead of the former, which means that the `accPendingYield` will be based on the entire position size instead of the amount of collateral being closed.

Next, in the call to `getTradeValuePure`,

```
function getTradeValuePure(
    uint collateral,
    int percentProfit,
    uint rolloverFee,
    uint accYield,
    int fundingFees,
    uint closingFee
) public view returns (uint, int, uint, uint) {
    // [...]

    int fees = pnl >= 0 ?
        int(rolloverFee) + int(closingFee) + fundingFees :
        int(rolloverFee) + int(closingFee) + fundingFees - int(accYield);

    int value = int(collateral) + pnl - fees;

    // [...]
}
```

This means that if the pnl of the position is positive, `accYield` directly affects `fees`, and so if it is incorrectly higher than it should be, the difference is given to the trader as assets during the close.

Impact

A trader can claim an unlimited amount of undue yield by opening a large position and then closing it in increments that are as small as possible. Each closure will credit the trader with a full amount of yield earned on the entire position despite only closing a token amount of collateral.

Recommendations

Pass `collateral` instead of `_trade.initialPosToken` into the call to `getPendingYield`.

Remediation

This issue has been acknowledged by Avantis Labs, Inc., and a fix was implemented in commit 659b495e ↗.

3.2. Market close callback trusts collateral amount

Target	TradingCallbacks		
Category	Coding Mistakes	**Severity**	Critical
Likelihood	Medium	**Impact**	Critical

Description

When a trader issues a market close, the amount of collateral to close is specified so that the trader has the option to not close all the collateral for a trade. The function in Trading checks that `_amount`, which is the amount of collateral closed, is less than the actual amount of collateral held by the contract for the trade `initialPosToken`:

```
function closeTradeMarket(
    uint _pairIndex,
    uint _index,
    uint _amount
) external payable onlyWhitelist whenNotPaused returns(uint orderId){
    // [...]
    uint coll = storageT.openTrades(msg.sender, _pairIndex, _index)
        .initialPosToken;
    // [...]
    require(_amount <= coll, "INV_AMOUNT");
    // [...]
}
```

However, the operator fulfills the order in a separate transaction, so things could have changed by then. In the callback that executes during this fulfillment, the amount of collateral to close is not checked against the total collateral. In `closeTradeMarketCallback`, `o.trade.initialPosToken` is actually the size of collateral to be closed, and `t.initialPosToken` is never read from.

Then, during `_unregisterTrade`, which is called with `_collateral` as the amount of collateral to close,

```
function _unregisterTrade(
    ITradingStorage.Trade memory _trade,
    int _percentProfit,
    uint _collateral,
    uint _feeAmountToken, // executor reward
    uint _lpFeeToken
) private returns (uint usdcSentToTrader) {
    // [...]
    (usdcSentToTrader, pnl, totalFees, lpYield) = pairInfos.getTradeValue(
        _trade,
        _collateral,
        _percentProfit,
        feeAfterRebate + _feeAmountToken
    );

    // [...]

    if ((_trade.initialPosToken == _collateral) ||
        (_collateral + totalFees >= _trade.initialPosToken)){
        storageT.unregisterTrade(_trade.trader, _trade.pairIndex,
            _trade.index);

        // [...]
    }
    else {
        pairInfos.storeTradeInitialAccFees(_trade.trader, _trade.pairIndex,
            _trade.index, _trade.buy);
        storageT.registerPartialTrade(_trade.trader, _trade.pairIndex,
            _trade.index, _collateral + totalFees);

        // [...]
    }
    // [...]
}
```

Note that during the attack, the `(_collateral + totalFees >= _trade.initialPosToken)` check in the conditional makes the logic consider the close as a complete close, rather than a partial close. So, the collateral amount isn't used to register any partial trade, so no underflow revert occurs.

Impact

An attacker can open a trade with a large amount of collateral, then issue a margin update to remove as much collateral as possible, and then issue a market close on the entire position. If we assume that the price does not change between the open and close, and the upkeeper fulfills the orders in the order they were issue, then all of the collateral removed during the margin update is free money because the market close on the entire position will close the original position size instead of the updated, smaller position. This allows the attacker to steal an unlimited amount of funds.

Recommendations

Check that the collateral being closed is less than the current amount of collateral in the callback.

Remediation

This issue has been acknowledged by Avantis Labs, Inc., and a fix was implemented in commit `efb4dcd2` ↗.

3.3. Failed SL update allocates fees incorrectly

Target	TradingCallbacks		
Category	Coding Mistakes	**Severity**	Critical
Likelihood	High	**Impact**	Critical

Description

When a trader wishes to update the SL on a trade whose pair has the SL guarantee enabled, an operator must supply the price first, to ensure that the SL guarantee is valid. During this process, in the operator's transaction, fees are deducted and accounted in `updateSlCallback`:

```
function updateSlCallback(uint orderId, uint price, uint spreadP)
    external override onlyPriceAggregator {
    // [...]

    ITradingStorage.Trade memory t = storageT.openTrades(o.trader,
        o.pairIndex, o.index);

    uint levPosUSDC = t.initialPosToken.mul(t.leverage);
    t.initialPosToken -= storageT.handleDevGovFees(t.trader, t.pairIndex,
        levPosUSDC >> 1, false, true, t.buy);

    if (
        price != 0 &&
        t.buy == o.buy &&
        t.openPrice == o.openPrice &&
        (t.buy ? o.newSl <= price : o.newSl >= price)
    ) {
        storageT.updateSl(o.trader, o.pairIndex, o.index,
            o.newSl, t.initialPosToken);
        emit SlUpdated(orderId, o.trader, o.pairIndex, o.index, o.newSl);
    }

    aggregator.unregisterPendingSlOrder(orderId);
}
```

However, the `handleDevGovFees` function assumes that the amount returned is always taken out of some quantity of assets, but the conditional means that the new `initalPosToken` is only saved inside `updateSl` if the new SL is valid.

The function `handleDevGovFees` is also where referrals are applied because it calls into `applyRe-`

`ferral`, which adds to the trader's referrer's reward balance if it exists.

Also, note that `updateSl` is where the SL timelock is reset, which means this process can be repeated immediately.

Impact

An attacker can steal an unlimited amount of assets in the form of referrer rewards by opening a position with a large OI and then making an unlimited number of invalid SL updates. Each update will transfer to the referrer an amount proportional to the leveraged position size, and each update does not change any state that would prevent the next update.

Recommendations

If fees are to be charged even if the SL update fails, then `updateSl` should always be called — so the conditional should change what SL is passed into it. Otherwise, the call to `handleDevGovFees` should be in the conditional.

Remediation

This issue has been acknowledged by Avantis Labs, Inc., and fixes were implemented in the following commits:

- `e543ec48` ↗
- `3d429d17` ↗
- `a3ce5fc2` ↗

3.4. Simultaneous `updateMargins` break leverage limits

Target	TradingCallbacks		
Category	Coding Mistakes	**Severity**	High
Likelihood	Low	**Impact**	High

Description

In order to do a margin update, a trader first calls `updateMargin` in Trading, which ensures that the update is within leverage limits and issues the margin update order:

```
function updateMargin(
    uint _pairIndex,
    uint _index,
    ITradingStorage.updateType _type,
    uint _amount
) external payable onlyWhitelist whenNotPaused returns(uint orderId){
    // [...]

    (t.leverage, t.initialPosToken) =
        PositionMath.calculateNewLeverage( /* [...] */ );

    require(
      t.leverage > 0 &&
       t.leverage >= aggregator.pairsStorage().pairMinLeverage(t.pairIndex) &&
       t.leverage <= aggregator.pairsStorage().pairMaxLeverage(t.pairIndex),
      "LEVERAGE_INCORRECT"
    );

    orderId = aggregator.getPrice(_pairIndex,
    IPriceAggregator.OrderType.UPDATE_MARGIN);
    aggregator.storePendingMarginUpdateOrder(
        orderId,
        IPriceAggregator.PendingMarginUpdate(msg.sender, _pairIndex, _index,
            _type, _amount, t.leverage)
    );

    emit MarginUpdateInitiated(msg.sender, t.pairIndex, orderId,
        block.timestamp);
}
```

Then, later, after a trusted operator supplies the price, the trade is fulfilled, which calls `updateMarginCallback` in TradingCallbacks. However, this function does not check the leverage; it only recalculates the rollover fees and new leverage and accepts it:

```
function updateMarginCallback(
    uint orderId, uint price, uint spreadP
) external override onlyPriceAggregator {
    // [...]

    (_trade.leverage, _trade.initialPosToken) =
        PositionMath.calculateNewLeverage( /* [...] */ );

    // [...] [check withdrawal threshold]
    // [...] [transfer funds accordingly]

    storageT.updateTrade(_trade);

    // [...]
}
```

Moreover, there is no concurrency protection that prevents a trader from submitting a second or third `updateMargin` order before the first one is fulfilled.

Impact

A trader can violate the minimum and maximum leverage for a position by calling `updateMargin` multiple times, where each time the margin update would not violate the limits if it was the only one that is applied.

Additionally, a similar issue exists in `updateSl` and `updateSlCallback`, with much less impact - if a user submits a second SL update before the first SL update is fulfilled, the second one can be fulfilled immediately afterwards even if it is during the limit timelock.

Recommendations

Check the leverage in `updateMarginCallback` as well as in `updateMargin`. Also, check the `slLastUpdated` in `updateSlCallback`.

Alternatively, add concurrency control so that only one `updateMargin`, or only one trade-modifying order (including partial-position closes and SL updates), can be outstanding at once.

Remediation

This issue has been acknowledged by Avantis Labs, Inc., and fixes were implemented in the following commits:

- `efb4dcd2` ↗
- `a3ce5fc2` ↗

3.5. Margin update misaligns group OI recordkeeping

Target	TradingCallbacks		
Category	Business Logic	**Severity**	High
Likelihood	High	**Impact**	High

Description

The PairStorage contract keeps track of the long and short open interest for each group of pairs to limit exposure to correlated assets. When a trade is opened, its OI is added to the group OI, and when it's closed its OI is removed.

During `updateMarginCallback`, the new position and leverage are calculated from the adjustment amount, and then if a margin fee was paid, the group OI is adjusted to reflect the change in position size:

```
function updateMarginCallback(
    uint orderId, uint price, uint spreadP
) external override onlyPriceAggregator {
    // [...]

    uint marginFees = pairInfos.getTradeRolloverFee(
        _trade.trader,
        _trade.pairIndex,
        _trade.index,
        _trade.buy,
        _trade.initialPosToken,
        _trade.leverage
    );

    (_trade.leverage, _trade.initialPosToken)
    = PositionMath.calculateNewLeverage(
        i.openInterestUSDC,
        _trade.initialPosToken,
        o._type ,
        o.amount,
        marginFees
    );

    // [...]

    if (marginFees != 0){
        storageT.vaultManager().allocateRewards(marginFees, false);
        storageT.priceAggregator().pairsStorage().updateGroupOI(
            _trade.pairIndex,
            marginFees.mul(o.oldLeverage),
            _trade.buy,
            false
        );
    }
    // [...]
}
```

The logic for `PositionMath.calculateNewLeverage` is here:

```
function calculateNewLeverage(
    uint _openInterestUSDC,
    uint _currentCollateral,
    ITradingStorage.updateType _type,
    uint _newAmount,
    uint _fees
) internal pure returns (uint newLeverage, uint newAmount) {
    if (_type == ITradingStorage.updateType.DEPOSIT) {
        newAmount = _currentCollateral + _newAmount - _fees;
        newLeverage = (_openInterestUSDC * _PRECISION) / (newAmount);
    } else if (_type == ITradingStorage.updateType.WITHDRAW) {
        newAmount = _currentCollateral - _newAmount - _fees;
        newLeverage = (_openInterestUSDC * _PRECISION) / (newAmount);
    }
}
```

However, note that, rounding errors aside, `calculateNewLeverage` ensures that `newLeverage * newAmount` is equal to the old leverage times amount, even with the margin fee taken out. So, the group OI does not actually change, so even if the margin fees are nonzero, `updateGroupOI` with the flag to reduce the group OI does not actually correspond to a decrease in OI.

Impact

Whenever `updateMargin` is called, the group OI risk parameter is permanently decreased by a small amount that is not recouped when the position is closed. This erroneously increases the risk that the protocol is willing to take on in a reproducible way.

Recommendations

Since the open interest does not change, `updateMargin` does not need to call `updateGroupOI`.

Remediation

This issue has been acknowledged by Avantis Labs, Inc., and a fix was implemented in commit `3d429d17` ↗.

3.6. Index can refer to other trade during fulfillment

Target	Trading		
Category	Coding Mistakes	**Severity**	High
Likelihood	Medium	**Impact**	High

Description

In Trading, all operations that both reference a trade and require a price for execution will refer to the trade by a three-tuple of its trader, pair index, and index. For example, when doing a market close, the trader calls `closeTradeMarket`, which then records a pending market close order:

```
function closeTradeMarket(
    uint _pairIndex,
    uint _index,
    uint _amount
) external payable onlyWhitelist whenNotPaused returns(uint orderId){
    // [...]

    storageT.storePendingMarketOrder(
        ITradingStorage.PendingMarketOrder(
            ITradingStorage.Trade(msg.sender, _pairIndex, _index, _amount,
                0, 0, false, 0, 0, 0, 0),
            0,
            0,
            0
        ),
        orderId,
        false
    );

    // [...]
}
```

However, inside the transaction in which the operator supplies the price, the index could refer to a completely different trade that was just opened. This is because the trade can be closed at any moment, and a new trade can also be opened at any moment, and when a new trade is created, the first empty index is reused:

```
function _registerTrade(ITradingStorage.Trade memory _trade)
    private returns (ITradingStorage.Trade memory) {
```

```
    // [...]

    _trade.index = storageT.firstEmptyTradeIndex(_trade.trader,
    _trade.pairIndex);

    // [...]

    return (_trade);
}
```

Impact

With this finding alone, the impact is that block builders can change what ends up happening by manipulating transaction ordering if multiple transactions are happening at once. For example, if a user submits a market close on an existing trade that is being liquidated at the same time, and then a market open, then the block builder can either let the market close fail to close any trade or apply the market close to the market open, causing unexpected behavior.

More importantly, this allows other findings to be exploited more easily because it provides an easy primitive for making the callback run on a different trade than what it seemed to Trading like it was created for. For example, it is not the only way to exploit Finding 3.2. ↗, but when that finding is exploited without this, the attacker does not have control over which — the margin update or the close — happens first, so sometimes it may fail. On the other hand, if the attacker creates a trade that would be closed (due to a take-profit limit, stop-loss limit, or liquidation), and then front-runs the closure, they can more reliably issue an order that will definitely point to a trade that is already gone.

Recommendations

Add concurrency controls so that a trade cannot have more than one order that would modify it to be pending execution, and then cancel that order if the trade is closed.

Alternatively, instead of using the first empty trade index for new trades, always assign a new identifier for each new trade.

Remediation

This issue has been acknowledged by Avantis Labs, Inc., and a fix was implemented in commit `8f4d309e` ↗.

3.7. Delegatable upgrade overlaps storage slots

Target	Delegatable		
Category	Coding Mistakes	**Severity**	High
Likelihood	High	**Impact**	High

Description

The Delegatable proxy-implementation upgrade would add a new base contract to the inheritance list for Trading:

```
pragma solidity 0.8.7;

contract Trading is PausableUpgradeable {
contract Trading is PausableUpgradeable, Delegatable {
    using PositionMath for uint;

    uint private constant _PRECISION = 1e10;
    uint private constant _MAX_SL_P = 80; // -80% PNL

    ITradingStorage public storageT;
    IPairInfos public pairInfos;
    IBlast public blast;

    // [...]
```

However, the Delegatable contract has some state of its own:

```
abstract contract Delegatable {
    mapping(address => address) public delegations;
    address internal _senderOverride;

    // [...]
```

Because of inheritance, before the upgrade, the Trading contract's proxy's account storage trie contains, starting from storage slot zero, the state variables of PauseableUpgradeable and then the state variables of Trading. This means that, after PauseableUpgradeable, the next state slots contain `storageT`, `pairInfos`, and `blast` in order. (The constants do not take up storage slots because the Solidity compiler inlines constants at compilation time.)

After the upgrade, since Trading now also inherits from Delegatable, the compiled storage layout will contain the state variables of PauseableUpgradeable, and then the state variables of Delegatable, and then the state variables of Trading. So, after the upgrade, the variable `delegations` will overlap with `storageT`, and `_senderOverride` will overlap with `pairInfos`, and so on.

Impact

It is not safe to just upgrade the implementation of Trading to add the Delegatable base contract because it shifts the storage layout of the Trading contract downwards. This scrambles most of the storage variables, including variables required for access control.

Recommendations

We recommend doing one of the following:

- Add a privileged external function to Trading that allows the sender to be set, and separately deploy Delegatable as a contract that can call that function.
- Make Delegatable have Trading as a base contract, and then point the proxy implementation to a deployment of Delegatable instead of Trading.
- Ensure Delegatable does not reserve any state slots by using assembly to set the slot of the state it needs to a location that does not collide with any Solidity-allocated storage slot.
- Deploy a new proxy and manually migrate the state.

Remediation

This issue has been acknowledged by Avantis Labs, Inc.. They plan to deploy a new proxy instead of doing a proxy-implementation upgrade.

3.8. The yield generated by Tranche cannot be claimed

Target	Tranche		
Category	Business Logic	**Severity**	High
Likelihood	High	**Impact**	High

Description

The idea behind the yield mode of the Tranche contract being `CLAIMABLE` was that the governance would call `claim` on the USDB address with recipient as the Tranche and then the governance would call the `claimYield` function on the Tranche contract to send the yield to the VaultManager. After reviewing the USDB source code, we found that only the contract that holds the token would be able to claim the yield; therefore, the governance could not claim the yield for the Tranche contract, leaving these yield stuck.

```
function claimYield() external onlyGov {
    uint claimableAmount = IERC20Rebasing(address(asset()))
        .getClaimableAmount(address(this));
    uint traderYield = claimableAmount*totalReserved/super.totalAssets();
    SafeERC20.safeTransfer(ERC20(asset()),
        address(vaultManager), traderYield);
    vaultManager.recieveBlastYield(traderYield);

    emit YieldTransferred(traderYield);
}
```

As shown above, the `claimYield` function does not call claim on the USDB token and expects the governance to call `claim` and transfer the funds to this contract.

Impact

The yield generated by the tokens in the Tranche contract could not be claimed.

Recommendations

We recommend calling the `claim` function on USDB in the `claimYield` function in Tranche so that the yield is transferred back to the Tranche and then to the VaultManager.

Remediation

This issue has been acknowledged by Avantis Labs, Inc., and a fix was implemented in commit `a7743cc4` ↗.

3.9. Generated yield might be stuck in the protocol

Target	TradingStorage, VaultManager, VeTranche		
Category	Business Logic	**Severity**	High
Likelihood	High	**Impact**	High

Description

On Blast, USDB is a rebasing token, and the default mode is automatic for both EOAs and contracts. This means that the USDB balance of EOAs and contracts will increase with time. In the contracts TradingStorage, VaultManager, and VeTranche, the default mode is automatic too. Therefore, any USDB in these contracts will earn yield and the balance of these contracts will increase with time.

Rebasing tokens might lead to some accounting errors in case the balance amount to be transferred out are stored as static balances in the contract's storage. For example, in the TradingStorage contract, the `govFeesUSDC` and `devFeesUSDC` are increased by the `govFees` and `feeAfterRebate - vaultAllocation - govFees` amount respectively in the `handleDevGovFees` function. But, when these fees are claimed by the governance, the actual amount to be transferred would be more than the stored value as these tokens also earned some yield and the balance of the TradingStorage contract would be increased by that amount.

```
function handleDevGovFees(
    address _trader,
    uint _pairIndex,
    uint _leveragedPositionSize,
    bool _usdc,
    bool _fullFee,
    bool _buy
) external override onlyTrading returns (uint feeAfterRebate) {
    uint fee = (
        _leveragedPositionSize * priceAggregator.openFeeP(_pairIndex)
    ) / _PRECISION / 100;

    if (!_fullFee) {
        fee /= 2;
    }

    (feeAfterRebate,) = applyReferral(_trader, fee, _leveragedPositionSize);

    uint vaultAllocation =
        (feeAfterRebate * (100 - _callbacks.vaultFeeP())) / 100;
    uint govFees = (feeAfterRebate * _callbacks.vaultFeeP()) / 100 >> 1;

    if (_usdc) IERC20Rebasing(usdc).safeTransfer(
        address(vaultManager), vaultAllocation);

    vaultManager.allocateRewards(vaultAllocation, false);
    govFeesUSDC += govFees;
    devFeesUSDC += feeAfterRebate - vaultAllocation - govFees;

    emit FeesCharged(_trader, _pairIndex, _buy, feeAfterRebate);
}
```

There is a similar issue with `rebates` as they are statically stored and the earned yield is not added to the referrer rebates. This would lead to the generated yield being stuck in the TradingStorage contract.

In VaultManager, the `pnlRewards` and `totalRewards` are increased with the static amount too and the generated yield from these rewards are used in the `_sendUSDCToTrader` function to be sent to the traders. Therefore, these extra yield rewards would never be distributed to the vault.

In case of VeTranche, the distributed rewards would earn some yield too, and thus would be inaccessible to the users and will be stuck in the contract.

Impact

The yield generated from the rebasing tokens would be stuck in the protocol.

Recommendations

We recommend using `CLAIMABLE` mode for yield and creating an external function that could call `claim` on the USDB rebasing token with the `recipient` as the VaultManager.

Remediation

This issue has been acknowledged by Avantis Labs, Inc., and a fix was implemented in commit a7743cc4 ↗.

3.10. Function `delegatedAction` should be payable

Target	Delegatable		
Category	Coding Mistakes	**Severity**	Medium
Likelihood	High	**Impact**	Medium

Description

The `delegatedAction` function in Delegatable is used to call functions in Trading on behalf of another account. It achieves this by having a `call_data` parameter and then doing `address(this).delegatecall` to pass in the calldata.

However, many functions in Trading, such as `updateMargin`, `openTrade`, `closeTradeMarket`, and others are payable and require payment in the form of ETH passed in with the transaction. On the other hand, `delegatedAction` is not payable:

```
function delegatedAction(address trader, bytes calldata call_data)
    external returns (bytes memory) {
```

Impact

The external functions in Trading that are payable and require a payment through `msg.value` will not be callable by `delegatedAction` because if value is sent in the call to `delegatedAction`, it will revert before executing function logic due to it itself not being payable.

Recommendations

Add the payable modifier to `delegatedAction`.

Remediation

This issue has been acknowledged by Avantis Labs, Inc., and a fix was implemented in commit `519dee0e` ↗.

3.11. Two-step ownership transfer, not two-step

Target	VaultManager		
Category	Coding Mistakes	**Severity**	Low
Likelihood	High	**Impact**	Low

Description

The two-step ownership transfer pattern is a safety pattern where, instead of the old owner of a smart contract setting the new owner, the old owner sets a pending new owner, and then the new owner must submit a transaction to accept ownership. This ensures that ownership cannot be lost due to transferring to the null address or another address that cannot accept ownership.

One of the changes in the audited code was implementing this two-step ownership transfer for the `gov` address for the Referral, TradingStorage, and VaultManager contracts. However, in all three contracts, this was how it was implemented:

```
modifier onlyGov() {
    require(msg.sender == gov);
    _;
}

function requestGov(address _gov) external onlyGov {
    require(_gov != address(0));
    requestedGov = _gov;
}

function setGov(address _gov) external onlyGov {
    require(_gov != address(0));
    require(_gov == requestedGov);
    // [...]
    gov = _gov;
}
```

Note that the `setGov` function is `onlyGov`, which means that it can only be called by the previous `gov` address, not the requested one.

Impact

Both `requestGov` and `setGov` must be called by the old governance to transfer governance, and governance may be transferred to an invalid address despite the two-step transfer pattern.

Recommendations

Change `setGov` to require that the sender be `requestedGov` and to not take any parameters so that it must be called by the `requestedGov` to accept ownership of the contract.

Remediation

This issue has been acknowledged by Avantis Labs, Inc., and a fix was implemented in commit `cc7734b6` ↗.

3.12. Execution of TP can be front-run for more profit

Target	TradingCallbacks		
Category	Business Logic	**Severity**	Low
Likelihood	Medium	**Impact**	Low

Description

During the execution of a limit-close order, the execution price is adjusted if the `guaranteedSlEn-abled` flag is set on the pair:

```
function executeLimitCloseOrderCallback(uint orderId, uint price,
    uint spreadP) external override onlyPriceAggregator {
    // [...]
    uint vPrice = aggregator.pairsStorage().guaranteedSlEnabled(t.pairIndex)
        ? o.orderType == ITradingStorage.LimitOrder.TP ? t.tp :
            o.orderType == ITradingStorage.LimitOrder.SL
            ? (t.buy && t.sl > t.openPrice) || (!t.buy && t.sl < t.openPrice)
                ? price
                : t.sl
            : price
        : price;

    // [...]
}
```

Breaking down this ternary, if `guaranteedSlEnabled` is true, then the price is set to the TP if the order type is TP. Since the TP can only be executed if the price is better (for the trader, depending on long/short) than the TP, the trader gets a worse execution price on pairs with guaranteed SL.

However, if the trader has not recently updated their TP so they are outside their TP timelock and they notice this transaction, they can set the TP to the execution first.

Impact

Trader TP limit closes execute at the same or worse price if the pair has guaranteed SL enabled. Also, traders can front-run this execution with a transaction, setting the TP to the future execution price to mitigate this effect.

Recommendations

Although the ability to race to adjust the TP upwards right before a TP execution is an intended side effect of how they are supposed to work, we recommend removing this price adjustment so that TP executions execute at the market price instead of the TP price, so that traders who are not sophisticated enough to monitor the mempool or self-execute their own TPs do not get penalized by this effect.

Remediation

This issue has been acknowledged by Avantis Labs, Inc.. Executing TPs at the TP instead of the price is an intentional design choice to protect the protocol in case the keeper bots are down.

3.13. Whitelist checks delegate address instead of user

Target	Trading		
Category	Business Logic	**Severity**	Low
Likelihood	Low	**Impact**	Low

Description

When the whitelist is turned on by governance, the whitelist only allows whitelisted accounts to use `onlyWhitelist` functions in Trading:

```
modifier onlyWhitelist() {
    if (isWhitelisted) require(whitelistedAddress[msg.sender],
    "WHITELIST_ONLY");
    _;
}
```

Most of the external functions, such as `openTrade` and `updateMargin`, are `onlyWhitelist`, so turning on the whitelist effectively restricts trading activity to whitelisted accounts.

However, when a function in Trading is called through `delegatedAction`, the account that is acted upon is the `_msgSender()` rather than `msg.sender`, which is the delegate.

Impact

If an account that is not whitelisted delegates to an account that is whitelisted, the nonwhitelisted account can open trades. Additionally, if an account is whitelisted and delegates to an account that is not, transactions will unexpectedly fail.

Recommendations

We recommend using `_msgSender()` in the modifier instead of `msg.sender` to more accurately check the sender when a delegation is occurring.

Remediation

This issue has been acknowledged by Avantis Labs, Inc., and a fix was implemented in commit `519dee0e` ↗.

3.14. LP yield is not reserved

Target	VaultManager		
Category	Coding Mistakes	Severity	Low
Likelihood	Low	Impact	Low

Description

When a trade is unregistered, the accumulated yield is added to the accLpYield variable in VaultManager. This yield is distributed back to the LPs. The function returnBlastYield could be called by the keeper to transfer the generated yield to the vault.

```
function returnBlastYield(uint _yield) external onlyKeeper{
    accLpYield -= _yield;
    storageT.usdc().transfer(address(vault), _yield);
    emit YieldReturned(traderYield);
}
```

The generated yield should be reserved in the VaultManager to be sent to the vault later, although this yield is not reserved in the _sendUSDCToTrader function when USDC is transferred to the trader.

```
function _sendUSDCToTrader(address _trader, uint _amount) internal {
    // For the extereme case of totalRewards exceeding vault Manager balance
    int256 balanceAvailable =
        int(storageT.usdc().balanceOf(address(this))) - int(totalRewards);
    if (int(_amount) > balanceAvailable) {
        // take difference (losses) from vaults
        uint256 difference = uint(int(_amount) - int(balanceAvailable));
        vault.withdrawAsVaultManager(difference);
    }

    require(storageT.usdc().transfer(_trader, _amount));
    emit USDCSentToTrader(_trader, _amount);
}
```

If the yield is transferred to the traders in the _sendUSDCToTrader call, the function returnBlastYield would fail as there would not be enough tokens in the VaultManager to cover the yield.

Impact

Calls to `returnBlastYield` might fail if the LP yield is not reserved.

Recommendations

We recommend reserving the `accLpYield` by subtracting it from the `balanceAvailable` in `_sendUS-DCToTrader`.

Remediation

This issue has been acknowledged by Avantis Labs, Inc., and a fix was implemented in commit 732855d5 ↗.

3.15. Operator fulfills revert if any one order fails

Target	Trading		
Category	Coding Mistakes	**Severity**	Informational
Likelihood	Low	**Impact**	Informational

Description

When an operator fulfills a sequence of orders, it calls `executeMarketOrders` with the below logic:

```
function executeMarketOrders(uint[] calldata orderId, uint256 price)
    external onlyOperator{
    for(uint i; i< orderId.length; ++i){
        storageT.priceAggregator().fulfill(orderId[i], price);
    }
}
```

However, the `fulfill` function will revert if the order is already fulfilled, instead of failing without a revert:

```
function fulfill(uint orderId, uint price)
    external payable override onlyTrading {
    Order storage r = orders[orderId];

    if (r.initiated) {
        // [...]
    }
    else {
        revert("ORDER_ALREADY_EXECUTED");
    }
}
```

Impact

This means that any operator can cause other operators to fail by fulfilling just one order in the transaction submitted by the other operator. There is not any impact because operators are trusted to not do that. This also means that operators must all coordinate to not fulfill the same transaction twice, or else waste gas.

Recommendations

Nevertheless, we recommend emitting an event or skipping the order instead of reverting if the order has already executed.

Remediation

This issue has been acknowledged by Avantis Labs, Inc.. The operators are trusted and assumed to not front-run each other or become desynchronized.

3.16. Full delegatee control of `call_data` is dangerous

Target	Trading		
Category	Business Logic	**Severity**	Informational
Likelihood	Low	**Impact**	Informational

Description

When an account sets another account as its delegate, that delegate is allowed to make calls on behalf of the delegatee (the first account) using `delegatedAction`:

```
function delegatedAction(address trader, bytes calldata call_data)
    external payable returns (bytes memory) {
    require(delegations[trader] == msg.sender, "DELEGATE_NOT_APPROVED");

    _senderOverride = trader;
    (bool success, bytes memory result) =
        address(this).delegatecall(call_data);
    if (!success) {
        // [...] (revert)
    }

    _senderOverride = address(0);

    return result;
}
```

However, if the delegatee has full control over the `call_data` variable when the delegate calls `delegatedAction` with the delegatee as the `trader`, they can encode a second call to `delegatedAction` in the `call_data`.

Since the function above checks that `delegations[trader] == msg.sender` instead of checking it to be `_msgSender()` (which would return `_senderOverride` if it is not null), instead of using the privileges of the delegatee, it will use the privileges of the delegate.

Impact

If the delegatee has full control over their `call_data`, then they can encode a second call to `delegatedAction` to call anything on behalf of anyone else delegated to the caller.

This finding is only of informational impact because Avantis Labs, Inc. assured us that the intended use of delegation is not to have multiple users delegate to the same central authority that authorizes the users off chain. It is intended for smart contract wallets to issue trades on behalf of their owner EOAs. If the delegate and the delegatee are agents of the same end user, and no delegate is shared between unrelated end users, then the finding has no impact.

Recommendations

We recommend making the first line of `delegatedAction` check `_msgSender()` instead of `msg.sender`.

Alternatively, another mitigation would be to make sure that the `call_data` is encoded in a secure, server-side manner so that unexpected function signatures (such as `delegatedAction`) cannot be issued by a delegate on behalf of a malicious delegatee.

Remediation

Avantis Labs, Inc. will ensure that delegation is not used in a way where multiple end users delegate to a central authority.

3.17. If governance is changed, new governance would not be able to claim gas

Target	Referral, TradingStorage		
Category	Business Logic	**Severity**	Informational
Likelihood	N/A	**Impact**	Informational

Description

On Blast, sequencer fees are redirected to dApps that induced them. Contracts that set gas mode to `Claimable` could claim the gas by calling the `claimAllGas` function. The entity that is allowed to claim the gas fees spent on a contract is known as its governor. By default, the governor of a smart contract is the contract itself. However, the governor could be changed by calling the `configureGovernor` function.

In the Bloom contracts, the governor of the contracts is set to the governance. If the governance is now changed, the new governance would not be able to call the `claimAllGas` function. It is important that the previous governance calls the configure function to change the governor of the contracts to the new governance address. If it is not called and the old governance access is lost, the new governance would not be able to claim the gas fee.

Impact

In certain scenarios after the governance address is changed, the gas-claiming functionality might be inaccessible to the new governance.

Recommendations

We recommend implementing the governor as a smart contract with the capability to change the governor to a new address if necessary. Ensure that these functions are access controlled, allowing only the governance to call them.

Remediation

This issue has been acknowledged by Avantis Labs, Inc., and a fix was implemented in commit 732855d5 ↗.

4. Discussion

The purpose of this section is to document miscellaneous observations that we made during the assessment. These discussion notes are not necessarily security related and do not convey that we are suggesting a code change.

4.1. Trader yield might be inaccessible for transfer to VaultManager

In the Tranche contract, the yield mode is set to claimable. Therefore, the governance would have to call `claimYield` in the tranche first and then call `claimYield` on the Blast contract with the recipient as the Tranche contract. There might be a scenario where there are not enough tokens in the tranche to cover the yield to be sent to the VaultManager, as yield is claimed after the call to `claimYield` on Tranche.

4.2. Incorrect argument in `YieldReturned` event

The function `returnBlastYield` in VaultManager emits an event `YieldReturned` with the argument as `traderYield`. But the actual yield returned back to the LPs is stored in the `_yield` variable.

```
function returnBlastYield(uint _yield) external onlyKeeper{
    accLpYield -= _yield;
    storageT.usdc().transfer(address(vault), _yield);
    emit YieldReturned(traderYield);
}
```

We recommend changing the argument to the `_yield` variable.

4.3. Comment-code mismatch for the `configureUSDBYield` function

The function `configureUSDBYield` is used to configure the yield mode of the contract, but the comments state that the function updates the USDC token contract address.

```
function configureUSDBYield(YieldMode _yieldMode) external onlyGov {
    require(address(usdc) != address(0));
    usdc.configure(_yieldMode);
}
```

We recommend modifying the comment to reflect similarity to what the code is doing.

4.4. Blast address does not need to be in storage

For this update, all the contracts were modified to save the Blast address in storage and configure the governor in the constructor or initializer:

```
blast = IBlast(_blast);
blast.configureClaimableGas();
blast.configureGovernor(storageT.gov());
```

However, for most contracts, the Blast address is not used anywhere else and is unlikely to change. So, this variable does not need to be in storage.

4.5. Confusing operator role naming in Trading

The Trading contract now has, in state, both an `address public operator` and a `mapping(address => bool) public isOperator`. These operators refer to two different things — the `operator` state address denotes the address that operator fees are paid to, whereas the `isOperator` mapping denotes who is authorized to provide the price. The `operator` can either be `isOperator` or not.

This is confusing because both roles are called "operator". We suggest renaming the fee-recipient variable name to prevent future confusion.

4.6. Market-close callback emits `MarketOpenCancelled`

In TradingCallbacks in `closeTradeMarketCallback`, if the price is unavailable or the trade is already closed, an event named `MarketOpenCancelled` is emitted. The naming of this event is confusing because a market close was actually cancelled.

We recommend either renaming the event `MarketCancelled` to share the same event between the market-open and market-close functions or adding a new event for `MarketCloseCancelled`.

5. Threat Model

This provides a full threat model description for various functions. As time permitted, we analyzed each function in the 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. Module: PairInfos.sol

Function: `setKeeper(address _keeper)`

Set the `keeper` address.

Inputs

- `_keeper`
    - **Control**: Fully controlled by the caller.
    - **Constraints**: None.
    - **Impact**: Address of the new keeper.

Branches and code coverage (including function calls)

Intended branches

- Updates the `keeper` address to the provided input.
    - ☑ Test coverage

Negative behavior

- Revert if `_keeper` is address(0).
    - ☐ Test coverage

Function: `setYieldRate(uint _rate)`

Sets the `yieldRate` variable to the provided input.

Inputs

- `_rate`
    - **Control**: Fully controlled by the caller.
    - **Constraints**: None.
    - **Impact**: The new yield rate to be set.

Branches and code coverage (including function calls)

Intended branches

- Sets the `yieldRate` as the provided input.
  - ☑ Test coverage

Negative behavior N/A.

5.2. Module: Trading.sol

Function: `cancelPendingMarketOrder(uint256 _id)`

Fallback method in case unregister is not working directly.

Inputs

- `_id`
  - **Control**: Fully controlled by the caller.
  - **Constraints**: None.
  - **Impact**: The ID for which the pending market order should be unregistered.

Branches and code coverage (including function calls)

Intended branches

- Calls `forceUnregisterPendingMarketOrder` function of the storage contract for the order ID.
  - ☑ Test coverage

Negative behavior N/A.

Function call analysis

- `storageT.forceUnregisterPendingMarketOrder(_id)`
  - **What is controllable?** `_id`.
  - **If the return value is controllable, how is it used and how can it go wrong?** No return value.
  - **What happens if it reverts, reenters or does other unusual control flow?** If it reverts, the entire call will revert — no reentrancy scenarios.

Function: `executeMarketOrders(uint[] calldata orderId, uint256 price)`

Keeper method to close pending market orders.

Inputs

- `orderId`
    - **Control**: Fully controlled by the caller.
    - **Constraints**: None.
    - **Impact**: The array of order IDs.
- `price`
    - **Control**: Fully controlled by the caller.
    - **Constraints**: None.
    - **Impact**: Pyth price.

Branches and code coverage (including function calls)

Intended branches

- Calls fulfill on all the order IDs.
    - ☑ Test coverage

Negative behavior N/A.

Function call analysis

- `storageT.priceAggregator().fulfill(orderId[i], price)`
    - **What is controllable?** `orderId` and `price`.
    - **If the return value is controllable, how is it used and how can it go wrong?** No return value.
    - **What happens if it reverts, reenters or does other unusual control flow?** If it reverts, the entire call will revert — no reentrancy scenarios.

Function: `removeWhitelist(address _address)`

Removes an address from the whitelist.

Inputs

- `_address`
    - **Control**: Fully controlled by the caller.
    - **Constraints**: None.
    - **Impact**: The address to remove from the whitelist.

Branches and code coverage (including function calls)

Intended branches

- Sets the `whitelistedAddress` for this address as false.
  - ☑ Test coverage

Negative behavior

- Revert if `_address` is address(0).
  - ☐ Test coverage

Function: `setMarketExecFeeReciever(address _reciever)`

Sets the new operator as the `_reciever`.

Inputs

- `_reciever`
  - **Control**: Fully controlled by the caller.
  - **Constraints**: None.
  - **Impact**: The address for the new operator to be set.

Branches and code coverage (including function calls)

Intended branches

- Updates the `operator` address.
  - ☑ Test coverage

Negative behavior

- Revert if `_reciever` is address(0).
  - ☐ Test coverage

Function: `updateOperator(address _operator, bool _isOperator)`

Updates the operator.

Inputs

- `_operator`
  - **Control**: Fully controlled by the caller.
  - **Constraints**: None.

    - **Impact**: The operator to update.
- `_isOperator`
    - **Control**: Fully controlled by the caller.
    - **Constraints**: None.
    - **Impact**: Sets the `isOperator` boolean to this value of the provided `_operator`.

Branches and code coverage (including function calls)

Intended branches

- Updates the `isOperator` mapping.
    - ☑ Test coverage

Negative behavior N/A.

6. Assessment Results

At the time of our assessment, the reviewed code was not deployed to the Blast L2 Mainnet.

During our assessment on the scoped Bloom Trading contracts, we discovered 17 findings. Three critical issues were found. Six were of high impact, one was of medium impact, four were of low impact, and the remaining findings were informational in nature. Avantis Labs, Inc. acknowledged all findings and implemented fixes.

6.1. Disclaimer

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 code added to the project after the version reviewed during 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 samples 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. These recommendations are not exhaustive, and we encourage our partners to consider them as a starting point for further discussion. We are happy to provide additional guidance and advice as needed.

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.