Zellic

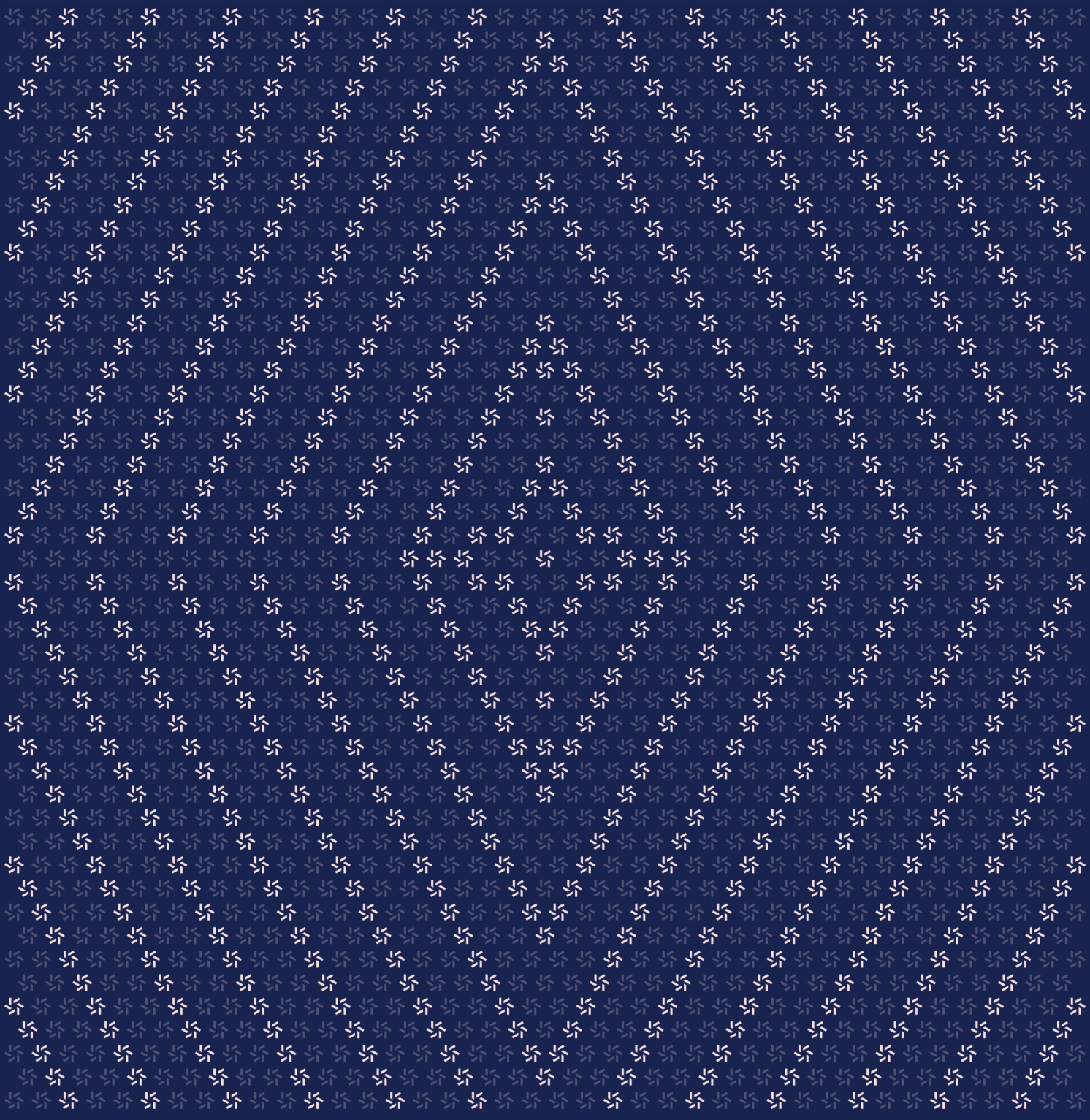
Prepared for
Blackhaven

Prepared by
Filipe Alves
Jinseo Kim
Zellic

January 19, 2026

Blackhaven (Core Contracts)

Smart Contract Security Assessment



Contents

About Zellic **4**

1. Overview **4**

1.1. Executive Summary 5

1.2. Goals of the Assessment 5

1.3. Non-goals and Limitations 5

1.4. Results 5

2. Introduction **6**

2.1. About Blackhaven (Core Contracts) 7

2.2. Methodology 7

2.3. Scope 9

2.4. Project Overview 9

2.5. Project Timeline 10

3. Detailed Findings **10**

3.1. Bond uses global `vestingTerm` instead of per-bond vesting 11

3.2. Missing oracle staleness validation 13

3.3. Oracle implementations must normalize to 1e18 decimals 15

3.4. BAM's `collectPremium` may fail due to insufficient liquidity 17

3.5. BAM's `collectPremium` vulnerable to sandwich attacks 19

3.6. LiquidityManager missing `onERC721Received` callback 21

3.7. RBTNote dust deposits can result in zero yield 22

3.8. RBTNote's `calculateSlash` precision loss allows penalty-free early exit near maturity 24

3.9. Unused `lastBacking` variable in BackingCalculator 26
3.10. Inconsistent decimal handling across codebase 28
3.11. No mechanism to remove backing tokens 30

4. Discussion 31
4.1. Bond debt decay behavior 32
4.2. The `addBacking` function's open access 32
4.3. Centralization risks 33

5. System Design 33
5.1. Component: Bond 34
5.2. Component: RBTNote 35
5.3. Component: BackingCalculator 36
5.4. Component: Backing Arbitrage Module (BAM) 37
5.5. Component: Minter 38
5.6. Component: RBT 38
5.7. Component: LiquidityManager 39

6. Assessment Results 39
6.1. Observations and recommendations 40
6.2. Disclaimer 41

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 Blackhaven from January 7th to January 14th, 2026. During this engagement, Zellic reviewed Blackhaven's core contracts' 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:

- Are the net-asset-value (NAV) and multiple-of-NAV (mNAV) calculations performed correctly, ensuring accurate backing ratios?
- Does the protocol correctly consume and use oracle data in its calculations?
- Does the protocol correctly enforce that bonds only mint when mNAV is greater than 1x?
- Could an attacker drain or manipulate reserve assets backing RBT?
- Are the premium capture mechanisms secure when market value exceeds backing value?

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
- Correctness of external oracle data feeds

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

1.4. Results

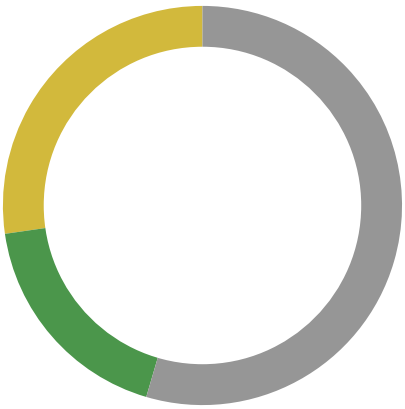
During our assessment on the scoped Blackhaven core contracts, we discovered 11 findings. No critical issues were found. Three findings were of medium impact, two were of low impact, and the remaining findings were informational in nature.

Additionally, Zellic recorded its notes and observations from the assessment for the benefit of

Blackhaven in the Discussion section (4. ↗).

Breakdown of Finding Impacts

Impact Level	Count
Critical	0
High	0
Medium	3
Low	2
Informational	6



2. Introduction

2.1. About Blackhaven (Core Contracts)

Blackhaven contributed the following description of Blackhaven (core contracts):

> BlackHaven is a decentralized protocol that issues RBT (Reserve Backed Token), a token backed by multiple reserve assets. The protocol maintains a 1:1 backing ratio and captures premium when the market value exceeds the backing value (mNAV > 1x).

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:

Blackhaven (Core Contracts)

Type	Solidity
Platform	EVM-compatible
Target	blackhaven-factory
Repository	https://github.com/blackhaven-xyz/blackhaven-factory ↗
Version	`5730ca6560ca203f90e3012e225a744c5ef42535`

Programs	`Bond.sol`
	`RBTNote.sol`
	`core/BackingCalculator.sol`
	`core/BAM.sol`
	`core/Minter.sol`
	`core/RBT.sol`
	`v3Liquidity/LiquidityManager.sol`

2.4. Project Overview

Zellic was contracted to perform a security assessment for a total of 1.7 person-weeks. The assessment was conducted by two consultants over the course of six calendar days.

Contact Information

The following project manager was associated with the engagement:

Pedro Moura
Engagement Manager
pedro@zellic.io ↗

The following consultants were engaged to conduct the assessment:

Filipe Alves
Engineer
filipe@zellic.io ↗

Jinseo Kim
Engineer
jinseo@zellic.io ↗

2.5. Project Timeline

The key dates of the engagement are detailed below.

January 7, 2026	Start of primary review period
January 8, 2026	Kick-off call
January 14, 2026	End of primary review period

3. Detailed Findings

3.1. Bond uses global `vestingTerm` instead of per-bond vesting

Target	Bond.sol		
Category	Business Logic	**Severity**	Medium
Likelihood	Medium	**Impact**	Medium

Description

The `redeem`, `redeemEarly`, and `getBondInfo` functions use `terms.vestingTerm` as the vesting term for all bonds:

```
function redeem(uint256 _tokenId) external nonReentrant {
    // [...]
    uint256 maturityTime = startTime + terms.vestingTerm;
    require(block.timestamp >= maturityTime, "Bond not fully vested");
    // [...]
}

function redeemEarly(uint256 _tokenId) external nonReentrant {
    // [...]
    uint256 maturityTime = startTime + terms.vestingTerm;
    // [...]
    uint256 vestedAmount = (payout * timeElapsed) / terms.vestingTerm;
    // [...]
}

function getBondInfo(uint256 _tokenId) external view returns (...) {
    // [...]
    vestingTerm = terms.vestingTerm;
    maturityTime = startTime + vestingTerm;
    // [...]
}
```

However, `terms.vestingTerm` can be updated by the owner via `setBondTerms(PARAMETER.VESTING, _input)`, and changing this value retroactively affects all existing bonds.

Impact

If the owner updates `vestingTerm` after bonds are created, existing bonds could mature earlier or later than users expected. Early redemption calculations in `redeemEarly` would also use incorrect

vesting periods, affecting the vested/unvested split and causing users to receive more or less RBT than they are entitled to based on their original bond terms.

Recommendations

Use the per-bond `vesting` field (stored in the `Bond` struct) instead of `terms.vestingTerm` in `redeem`, `redeemEarly`, and `getBondInfo` — for example, in `redeem`:

```
function redeem(uint256 _tokenId) external nonReentrant {
    Bond memory info = bonds[_tokenId];
    uint256 maturityTime = info.lastBlockTimestamp + info.vesting;
    require(block.timestamp >= maturityTime, "Bond not fully vested");
    // [...]
}
```

The same pattern should be applied to `redeemEarly` and `getBondInfo`.

Remediation

This issue has been acknowledged by Blackhaven, and a fix was implemented in commit `7a9b7526` ↗.

3.2. Missing oracle staleness validation

Target	BackingCalculator.sol		
Category	Business Logic	**Severity**	High
Likelihood	Low	**Impact**	Medium

Description

The `singleNAV` and `FDV` functions in BackingCalculator consume oracle prices through `IOracle.getPrice()` without any staleness checks or validation:

```
function singleNAV(address _token)
    public view returns (uint256 _netAssetValue) {
    BackingTokenDetails memory backingTokenDetails
    = backingTokenDetailsForAddress[_token];
    require(backingTokenDetails.oracle != address(0), "Oracle not set for
    token");
    return IERC20(_token).balanceOf(backingStorage)
    * IOracle(backingTokenDetails.oracle).getPrice() / 1e18;
}

function FDV() public view returns (uint256 _FDV) {
    require(backedTokenOracle != address(0), "Backed token oracle not set");
    return IERC20(backedToken).totalSupply()
    * IOracle(backedTokenOracle).getPrice() / 1e18;
}
```

These values are used in critical calculations: `NAV()` aggregates all `singleNAV` values, `mNAV()` uses both `NAV()` and `FDV()` to calculate the premium multiple, Bond pricing uses `NAV()` to enforce the minimum price floor, and BAM uses `mNAV()` to determine premium capture rates.

Impact

If an oracle becomes unresponsive or returns stale data, bond pricing could use incorrect minimum price floors, allowing bonds to be sold below intrinsic value. BAM premium capture calculations could also be incorrect, capturing too much or too little premium. Users could be adversely affected by trades executed against stale prices.

Recommendations

Ensure the oracle wrapper implementation includes staleness checks to validate that price data is fresh before use. Additionally, define fallback behavior for when oracles are unresponsive (e.g., pause bond sales, use the last known good price with time decay).

Remediation

This issue has been acknowledged by Blackhaven. Blackhaven notified us that they plan to use oracles (or wrappers of them) whose `getPrice` function reverts when the price becomes stale or zero, and they stated this in the documentation in commit `fd7f5a3a` ↗.

3.3. Oracle implementations must normalize to 1e18 decimals

Target	BackingCalculator.sol		
Category	Business Logic	**Severity**	High
Likelihood	Low	**Impact**	Medium

Description

BackingCalculator assumes all oracle implementations return prices scaled to 1e18:

```
function singleNAV(address _token)
    public view returns (uint256 _netAssetValue) {
    // [...]
    return IERC20(_token).balanceOf(backingStorage)
    * IOracle(backingTokenDetails.oracle).getPrice() / 1e18;
}

function FDV() public view returns (uint256 _FDV) {
    // [...]
    return IERC20(backedToken).totalSupply()
    * IOracle(backedTokenOracle).getPrice() / 1e18;
}
```

However, native price feeds from providers like Redstone often use different decimal precision. For example, Redstone's USDC/USD feed returns eight decimals, not 18. Additionally, the calculation assumes backing tokens (currently USDM and MEGA) use 18 decimals. If a backing token with different decimals is added, the NAV calculation would be incorrect.

Impact

If an oracle wrapper does not properly normalize the price to 1e18 before returning from `getPrice()`, NAV calculations would be off by orders of magnitude (e.g., 10^{10} for eight-decimal feeds), bond minimum price floors would be incorrect, and users could purchase bonds at drastically incorrect prices.

Recommendations

Document the 1e18 price scaling requirement clearly in the `IOracle` interface, and ensure oracle wrappers normalize prices from native feeds to 1e18 before returning. For backing tokens, either restrict additions to 18-decimal tokens or update `singleNAV` to normalize token balances using

`IERC20Metadata(_token).decimals()`. Additionally, add deployment tests that verify oracle prices are in expected ranges and backing token decimals are as expected.

Remediation

This issue has been acknowledged by Blackhaven. Blackhaven notified us that they plan to use oracles (or wrappers of them) that return prices scaled to 18 decimals, and they stated this in the documentation in commit `8f4dcb6c` ↗. The oracle wrapper implementation and the correctness of external oracle data feeds are outside the scope of this audit.

3.4. BAM's `collectPremium` may fail due to insufficient liquidity

Target	BAM.sol		
Category	Protocol Risks	**Severity**	Low
Likelihood	Low	**Impact**	Low

Description

The `collectPremium` function swaps RBT for USDM via Uniswap V3 with a slippage tolerance (default 5%):

```
function _swapToUSDM(uint256 _amount) internal {
    // [...]
    if (rbtOracle != address(0)) {
        uint256 rbtPrice = IOracle(rbtOracle).getPrice();
        uint256 expectedUSDM = (_amount * rbtPrice) / 1e18;
        amountOutMinimum = (expectedUSDM * (BPS_DENOM - slippageToleranceBps))
    / BPS_DENOM;
    }
    // [...]
    uint256 amountOut = swapRouter.exactInputSingle(params);
    // [...]
}
```

If the Uniswap pool lacks sufficient depth to execute the swap within the slippage tolerance, the transaction reverts and premium capture fails entirely. This couples the protocol's premium mechanism to pool liquidity conditions.

Impact

Premium capture becomes unavailable during periods of low liquidity, meaning RBT accumulated in the BAM contract cannot be converted to backing. This compromises the protocol's ability to strengthen backing during high mNAV periods. However, the practical risk is limited since the majority of liquidity will be protocol-owned, swap sizes are inherently constrained by `maxSupplyPercentBps`, and the slippage tolerance can be adjusted via `updateSlippageToleranceBps()`.

Recommendations

Consider refactoring the swap process in order to have the slippage tolerance and/or the swap size dynamically adjusted based on the pool's depth. This could be done by using `sqrtPriceLimitX96` for example. Alternatively, consider monitoring pool depth relative to expected swap sizes. If liquidity issues arise, the slippage tolerance can be adjusted via `updateSlippageToleranceBps()`. As a suggestion, a dynamic slippage mechanism based on swap size could be useful — it would tighten protection for smaller collections without impacting larger ones.

Remediation

This issue has been acknowledged by Blackhaven.

3.5. BAM's `collectPremium` vulnerable to sandwich attacks

Target	BAM.sol		
Category	Protocol Risks	**Severity**	Low
Likelihood	Medium	**Impact**	Low

Description

The `collectPremium` function in BAM swaps RBT for USDM via Uniswap V3 with a configurable slippage tolerance (default 5%):

```
function _swapToUSDM(uint256 _amount) internal {
    // [...]
    if (rbtOracle != address(0)) {
        uint256 rbtPrice = IOracle(rbtOracle).getPrice();
        uint256 expectedUSDM = (_amount * rbtPrice) / 1e18;
        amountOutMinimum = (expectedUSDM * (BPS_DENOM - slippageToleranceBps))
    / BPS_DENOM;
    }
    // [...]
}
```

This creates an opportunity for sandwich attacks; an attacker front-runs `collectPremium` by buying RBT (pushing the price up), `collectPremium` executes at a worse price within the slippage tolerance, and the attacker back-runs by selling RBT (profiting from the price impact). The attack is profitable when the MEV extracted exceeds gas costs. With 5% slippage tolerance, attackers can extract up to 5% of the swap value.

Impact

This could result in value leakage from the protocol during premium collection swaps, reduced backing accumulation efficiency, and MEV bots systematically extracting value. The practical impact is limited since swap sizes are capped by `maxSupplyPercentBps` (default 1% of supply) and slippage can be adjusted.

Recommendations

Slippage protection is already calculated based on the oracle feed, and premium collection swap sizes are inherently limited by `maxSupplyPercentBps`. The slippage tolerance can be reduced from the initial 5% if MEV extraction becomes a concern. Monitor swap execution, and adjust

`slippageToleranceBps` as needed based on observed MEV activity.

Remediation

This issue has been acknowledged by Blackhaven.

3.6. LiquidityManager missing `onERC721Received` callback

Target	LiquidityManager.sol		
Category	Business Logic	**Severity**	Informational
Likelihood	N/A	**Impact**	Informational

Description

The LiquidityManager contract is designed to hold and manage Uniswap V3 LP position NFTs. However, it does not implement the `onERC721Received` callback function required by the ERC-721 standard for contracts to receive NFTs via `safeTransferFrom`.

Without this callback, any attempt to transfer an LP position NFT to the contract using `safeTransferFrom` (the recommended method) will revert.

Impact

LP position NFTs cannot be transferred to the contract using `safeTransferFrom`, requiring the use of `transferFrom` as a workaround. This does not follow ERC-721 best practices for contract receivers.

Recommendations

Implement the `IERC721Receiver` interface by adding the `onERC721Received` callback function. This follows ERC-721 best practices and prevents potential issues with integrations that default to `safeTransferFrom`.

Remediation

This issue has been acknowledged by Blackhaven, and a fix was implemented in commit `0d1f25ce` ↗.

3.7. RBTNote dust deposits can result in zero yield

Target	RBTNote.sol		
Category	Business Logic	**Severity**	Informational
Likelihood	N/A	**Impact**	Informational

Description

The `calculateYield` function uses integer division, which can result in zero yield for dust deposits:

```
function calculateYield(uint256 _principal, uint256 _termDuration)
    public pure returns (uint256 yieldAmount) {
    uint256 yieldBps = calculateYieldRate(_termDuration);
    return (_principal * yieldBps) / BPS_DENOM;
}
```

For a two-week term (the minimum and worst case), `yieldBps` is approximately 78 basis points, meaning deposits below ~129 WEI would result in zero yield due to integer truncation. This creates notes where users lock RBT but receive nothing at maturity.

Impact

The practical impact is minimal given the negligible amounts involved, but it creates an accounting inconsistency where `totalYield` could be zero for a valid note and does not align with user expectations.

Recommendations

Add a minimum deposit check to ensure nonzero yield:

```
function deposit(uint256 _amount, uint256 _termDuration)
    external nonReentrant returns (uint256 tokenId) {
    require(_amount > 0, "Amount must be > 0");
    require(_termDuration >= MIN_TERM, "Term too short");
    require(_termDuration <= MAX_TERM, "Term too long");

    uint256 yieldAmount = calculateYield(_amount, _termDuration);
    require(yieldAmount > 0, "Deposit too small for yield");
```

```
    // Check sufficient yield available
    require(yieldAmount <= availableYield(), "Insufficient yield available");
    // [...]
}
```

Remediation

This issue has been acknowledged by Blackhaven, and a fix was implemented in commit 48e46f5a ↗.

3.8. RBTNote's `calculateSlash` precision loss allows penalty-free early exit near maturity

Target	RBTNote.sol		
Category	Business Logic	**Severity**	Informational
Likelihood	N/A	**Impact**	Informational

Description

The `calculateSlash` function has precision loss that allows users to exit very close to maturity without paying the early redemption penalty:

```
function calculateSlash(uint256 _tokenId)
    public view returns (uint256 slashAmount) {
    // [...]
    uint256 timeRemaining = note.maturityTime - block.timestamp;
    uint256 percentRemaining = (timeRemaining * BPS_DENOM)
    / note.termDuration;

    uint256 slashBps = (MAX_SLASH_BPS * percentRemaining) / BPS_DENOM;
    return (note.principal * slashBps) / BPS_DENOM;
}
```

When `timeRemaining * BPS_DENOM < note.termDuration`, `percentRemaining` rounds down to zero, resulting in zero slash. This allows **two-week terms** — exiting ~2 minutes early with no penalty (`timeRemaining < 120.96 seconds`) — and **52-week terms** — exiting ~52 minutes early with no penalty (`timeRemaining < 3,144.96 seconds`).

Impact

Users could technically bypass the early redemption slash by timing their exit very precisely near maturity. However, early redemption forfeits all yield, which would be substantial at 99%+ vesting, making this behavior economically irrational. Users would be better off waiting the remaining minutes to receive full principal plus yield. The protocol also benefits if users exit early since the forfeited yield returns to the pool. Therefore, the economic impact is effectively zero, though this precision loss may represent unintended behavior worth addressing for consistency.

Recommendations

Consider implementing a minimum slash to ensure the penalty mechanism is enforced consistently, even for edge cases near maturity:

```
function calculateSlash(uint256 _tokenId)
    public view returns (uint256 slashAmount) {
    // [...]
    // Slash scales from 10% at 100% remaining to 0% at 0% remaining
    uint256 slashBps = (MAX_SLASH_BPS * percentRemaining) / BPS_DENOM;
   if (slashBps == 0) {
       slashBps = 1;
   }

    return (note.principal * slashBps) / BPS_DENOM;
}
```

Remediation

This issue has been acknowledged by Blackhaven.

3.9. Unused `lastBacking` variable in BackingCalculator

Target	BackingCalculator.sol		
Category	Coding Mistakes	**Severity**	Informational
Likelihood	N/A	**Impact**	Informational

Description

The `addBacking` function stores a `lastBacking` value in the `BackingTokenDetails` struct, but this value is never used anywhere. All functions that need backing amounts (`backingPerToken`, `singleNAV`) read directly from the current token balance via `balanceOf(backingStorage)` instead:

```
function addBacking(address _backingToken, uint256 _amount) external {
    // [...]
    backingTokenDetails.lastBacking = IERC20(_backingToken).balanceOf(
        backingStorage);
    backingTokenDetailsForAddress[_backingToken] = backingTokenDetails;
    // [...]
}

function backingPerToken(address _backingToken)
    public view returns (uint256 _backingPerToken) {
    uint256 totalSupply = IERC20(backedToken).totalSupply();
    if (totalSupply == 0) return 0;
    return 1e18 * IERC20(_backingToken).balanceOf(backingStorage)
    / totalSupply;
}
```

Impact

This results in a minor gas cost for unused storage writes and reduces code clarity, as readers may wonder about the purpose of the field. There is no security or functional impact.

Recommendations

Consider removing the `lastBacking` field from `BackingTokenDetails` and the assignment in `addBacking`, or document its purpose if intended for future use (e.g., historical tracking via events).

Remediation

This issue has been acknowledged by Blackhaven, and a fix was implemented in commit b253a713 ↗.

3.10. Inconsistent decimal handling across codebase

Target	Multiple contracts		
Category	Coding Mistakes	**Severity**	Informational
Likelihood	N/A	**Impact**	Informational

Description

The codebase inconsistently handles RBT decimals. Some code hardcodes `1e18`,

```
// Bond.sol
require(payout >= 10 ** IERC20Metadata(address(RBT)).decimals() / 100, "Bond
    too small");
```

while other code reads decimals dynamically:

```
// Bond.sol
function valueOfToken(address _token, uint256 _amount)
    public view returns (uint256 value_) {
    value_ = (_amount * 10 ** IERC20Metadata(address(RBT)).decimals())
    / 10 ** IERC20Metadata(_token).decimals();
}
```

Impact

There is no functional impact since RBT uses 18 decimals; both approaches are functionally equivalent. However, the inconsistency reduces code readability and maintainability and could lead to bugs if a developer assumes one convention and encounters the other.

Recommendations

Standardize decimal handling across the codebase by always using `IERC20Metadata(token).decimals()` for flexibility. This approach ensures correctness regardless of token decimal configuration and makes the code more maintainable.

Remediation

This issue has been acknowledged by Blackhaven, and a fix was implemented in commit 4322498e ↗.

3.11. No mechanism to remove backing tokens

Target	BackingCalculator.sol		
Category	Business Logic	**Severity**	Low
Likelihood	Low	**Impact**	Informational

Description

Once a backing token is added via `addBackingToken`, there is no way to remove it from the `backingTokens` array or update its oracle:

```
function addBackingToken(address _token, address _oracle) external onlyOwner {
    require(!backingTokenDetailsForAddress[_token].isBackingToken, "Already
    Backing Token");
    backingTokenDetailsForAddress[_token].isBackingToken = true;
    backingTokenDetailsForAddress[_token].oracle = _oracle;
    backingTokens.push(_token);

    emit BackingTokenAdded(_token, _oracle);
}
```

If a backing token needs to be deprecated or its oracle implementation needs to be changed,

- the oracle address could be updated via a proxy pattern (if the oracle is upgradable);
- removing a token entirely requires deploying a new BackingCalculator contract; and
- this cascades to redeploying Bond (immutable `backingCalculator` reference) and updating BAM.

Impact

This limits upgradability for backing token management. If a backing token needs to be removed, the upgrade path is complex; it requires deploying a new BackingCalculator contract, which cascades to redeploying Bond (due to its immutable `backingCalculator` reference) and updating BAM. However, the practical impact is limited since the Blackhaven team only plans to support USDM and MEGA as backing tokens, reducing the likelihood of needing removal functionality.

Recommendations

Consider adding a `removeBackingToken` function that sets `isBackingToken` to false (effectively disabling the token without array removal), and an `updateBackingTokenOracle` function to allow

oracle updates without redeployment.

Remediation

This issue has been acknowledged by Blackhaven.

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. Bond debt decay behavior

The `deposit` function calls `decayDebt()` before adding new debt, which resets the decay baseline:

```
function decayDebt() internal {
    totalDebt = totalDebt - debtDecay();
    lastDecay = block.timestamp;
}
```

This creates a specific behavior where multiple deposits during a vesting term cause debt to decay at an effective exponential rate (~1/e per vesting term) rather than linearly. For example, take a two-month vesting term. If a bond of 1 RBT is created at t = 0, then another at t = 1 month, the debt at t = 2 months would be 0.75 RBT rather than the 0.5 RBT expected from pure linear decay (because the second deposit reset `lastDecay`, causing the first bond's remaining debt to decay slower).

Blackhaven notified us that this behavior is intended. Debt decays continuously as a pricing mechanism rather than strict per-bond accounting. When bonds are redeemed, debt is not explicitly reduced; it decays naturally over time. The `currentDebt()` function reflects effective debt after decay, which drives bond pricing through `debtRatio()`.

4.2. The `addBacking` function's open access

The `addBacking` function in BackingCalculator has no access controls. Blackhaven notified us that this is intentional; they explained that 1) the function may only add backing to the system (never remove it), 2) the function requires the caller to hold and approve the backing tokens, and 3) anyone adding backing is beneficial to the protocol. We agree that this does not pose a risk to the protocol.

However, off-chain infrastructure should be aware that `BackingAdded` events can be emitted by any address, not just trusted protocol contracts. This has no security implications (adding backing only increases NAV), but monitoring systems should account for this when attributing event sources.

4.3. Centralization risks

The protocol has several centralization points users should be aware of. Backing tokens are held at a separate `backingStorage` address rather than within the contract, meaning their security depends on that address's configuration (likely a multi-sig). The LiquidityManager owner can retrieve LP position NFTs and any tokens via `retrieveNFP` and `retrieveStuckToken`. The Minter owner controls which addresses can mint RBT tokens — a malicious (possibly compromised) owner could whitelist an arbitrary address to mint unlimited RBT, diluting existing holders. Additionally, bond terms, BAM parameters, and oracle addresses are all owner-controlled.

These are standard patterns for DeFi protocols providing operational flexibility. Consider documenting the expected `backingStorage` configuration, governance processes for parameter changes, and emergency procedures.

5. System Design

This provides a description of the high-level components of the system and how they interact, including details like a function's externally controllable inputs and how an attacker could leverage each input to cause harm or which invariants or constraints of the system are critical and must always be upheld.

Not all components in the audit scope may have been modeled. The absence of a component in this section does not necessarily suggest that it is safe.

5.1. Component: Bond

Description

The RBTBonding contract allows users to bond USDM for discounted RBT with vesting. It uses a control variable (BCV) to dynamically price bonds based on the debt ratio, and each bond is minted as an ERC-721 NFT. The contract implements a premium capture mechanism that distributes excess RBT minted (based on mNAV) across multiple destinations: BAM, protocol-owned liquidity (POL), the team wallet, and the reward wallet.

The Bond contract is responsible for the following:

- **Bond creation.** Users deposit USDM via `deposit` and receive discounted RBT that vests over a configurable term. The bond price is calculated from the control variable and debt ratio, with a minimum price enforced at 1x mNAV.
- **Vesting and redemption.** Bonds vest linearly over the vesting term. Users can `redeem` fully vested bonds or `redeemEarly` to receive the vested portion (forfeiting the unvested amount to the reward wallet).
- **RBT Note discount.** Users holding qualifying RBT Notes (term > 12 weeks, principal ≥ 1,000 RBT) receive a bonus discount of up to 2.5% on bond prices.
- **Premium distribution.** When bonds are sold above 1.05x mNAV, extra RBT is minted and distributed: 25% to BAM, 25% to POL (via LiquidityManager), 10% to the team wallet, and 40% to the reward wallet (configurable via BPS).
- **Debt management.** Total debt decays linearly over the vesting term, affecting bond pricing. The control variable can be adjusted incrementally toward a target.

Invariants

Bond pricing

- The bond price must not fall below the minimum price (NAV / `totalSupply`, i.e., 1x mNAV).
- The bond price is calculated as `(controlVariable * debtRatio) / 1e11`, subject to the minimum floor.
- The bond-price discount for RBT Note holders must not exceed `MAX_NOTE_DISCOUNT_BPS` (250 bps / 2.5%).

Payout limits

- A single bond payout must be at least 0.01 RBT (1e16 WEI).
- A single bond payout must not exceed `maxPayout` (`totalSupply × maxPayout / 100000`).
- The total debt after a deposit must not exceed `terms.maxDebt`.

Vesting integrity

- The `redeem` function can only be called when `block.timestamp >= maturityTime` (fully vested).
- The `redeemEarly` function can only be called when `block.timestamp < maturityTime`.
- Forfeited (unvested) bond amounts must always be transferred to `rewardWallet`, never burned or lost.

Distribution BPS

- Together, `premiumCaptureBps + polBps + teamWalletBps + rewardWalletBps` must equal `BPS_DENOM` (10,000).

Control variable adjustment

- The adjustment rate must not exceed 3% of the current BCV (`_increment <= controlVariable * 30 / 1000`).
- Adjustments can only occur after the buffer period has elapsed.

Attack surface

- **RBTNote discount calculation.** Discount calculation queries the external `RBTNote` contract and iterates through the user's notes.
- **Oracle dependency.** It relies on `BackingCalculator.NAV()` for the minimum price floor. Stale or manipulated oracle prices affect bond pricing.
- **LiquidityManager integration.** This includes external calls to LiquidityManager for POL distribution.

5.2. Component: RBTNote

Description

The RBTNote contract is a fixed-term RBT deposit system where users lock RBT for a specified duration (2–52 weeks) and receive yield at maturity. Each deposit is represented as an ERC-721 NFT, enabling transferability. The yield formula `Yield = 19% * (t/52) + 33% * (t/52)^2` incentivizes longer lock periods with a quadratic bonus. Early redemption incurs a slash penalty (up to 10%) and forfeits all yield.

Invariants

Yield availability

- New deposits can only be created if `yieldAmount <= availableYield()`.
- Also, `availableYield() = balance - totalPrincipal - totalYield`.

Term bounds

- Term duration must be `>= MIN_TERM` (two weeks) and `<= MAX_TERM` (52 weeks).

Redemption rules

- The `redeem` function requires `block.timestamp >= note.maturityTime`.
- The `redeemEarly` function requires `block.timestamp < note.maturityTime`.
- Only the NFT owner can redeem their note.

Attack surface

- **Yield availability dependency.** Deposits require sufficient `availableYield()`. If contract balance is depleted, new deposits revert.
- **RBT token transfers.** External RBT transfers are via `safeTransferFrom` for deposits and `safeTransfer` for redemptions.
- **Precision loss in calculations.** This includes integer division in yield and slash calculations.

5.3. Component: BackingCalculator

Description

The BackingCalculator contract computes NAV and mNAV for RBT based on backing tokens held at a designated `backingStorage` address. NAV is the sum of `balance * oraclePrice` for each registered backing token, while mNAV is `FDV / NAV`, indicating premium/discount. The contract serves as the central source of truth for backing-related calculations used by Bond and BAM.

Invariants

Backing-token uniqueness

- Once registered, a token cannot be added again.

Registration permanence

- There is no function to remove a backing token once added.

Attack surface

- **Oracle dependency.** NAV and mNAV depend entirely on oracle prices. This affects all dependent contracts (Bond pricing, BAM premium capture).
- **Backing storage address.** All backing tokens are held at `backingStorage`. Security depends on that address's configuration.

5.4. Component: Backing Arbitrage Module (BAM)

Description

The Backing Arbitrage Module (BAM) captures premium from RBT when mNAV exceeds a threshold and converts it to USDM to add to the protocol's backing. This mechanism helps maintain price stability by selling RBT when trading above intrinsic value. Premium is collected periodically via `collectPremium()` and swapped to USDM via Uniswap V3. The premium rate scales linearly with mNAV above `MIN_MNAV`, and swap amounts are capped by `maxSupplyPercentBps` to limit price impact.

Invariants

Premium activation

- Premium capture only occurs when `mNAV > MIN_MNAV`.
- Also, `MIN_MNAV` is immutable and must be `> 1e18` (enforced at construction).

Collection timing

- The `collectPremium` function can only execute when `block.timestamp >= nextPremiumCollection`.

Amount caps

- The premium amount is capped by `maxSupplyPercentBps` (as a percentage of the total supply).
- The actual swap amount is limited to the contract's RBT balance.

Attack surface

- **Oracle-dependent slippage protection.** Slippage protection relies on `backedTokenOracle`. If misconfigured, manipulated, or unset (`address(0)`), swaps could execute at unfavorable prices.
- **External `collectPremium` call.** Anyone can call `collectPremium` after the time elapses, exposing swaps to MEV extraction.
- **The `poolFee` mismatch.** The pool fee is hardcoded to 10,000 (1%). If the actual Uniswap pool uses a different fee tier, swaps will fail.

5.5. Component: Minter

Description

The Minter contract manages the whitelist of addresses authorized to mint RBT tokens. It acts as an intermediary between authorized contracts (e.g., Bond) and the RBT token, providing a centralized point of control for minting permissions. Whitelisted addresses call `mint(to, amount)`, which forwards the request to `RBT.mint()`.

Invariants

Access control

- Only addresses in the `allowedMinters` mapping can call `mint`.
- Only the owner can modify the `allowedMinters` mapping.

Attack surface

- **Centralized mint control.** The owner has full control over which addresses can mint. A compromised owner could whitelist malicious contracts for unlimited minting.

5.6. Component: RBT

Description

The RBT contract is the core ERC-20 token for the protocol. It extends OpenZeppelin's ERC20 and Ownable, adding controlled minting (via a designated `minter` address, typically set to the Minter contract) and public burning.

Invariants

Minting access

- Only the `minter` address can call `mint`.

Minter control

- Only the owner can call `updateMinter`.

Attack surface

- **Centralized minter control.** If the `minter` address is compromised or set incorrectly, minting functionality is broken or exploited. The owner controls minter updates via `updateMinter`.

5.7. Component: LiquidityManager

Description

The LiquidityManager contract is a thin wrapper around Uniswap V3's NonfungiblePositionManager for managing POL. It holds the LP position NFT and provides functions to increase liquidity via `increaseLiquidityCurrentRange()` and collect fees via `collectAllFees()`.

Invariants

Fee collection access

- Only `feeAddress` can call `collectAllFees`.

6. Assessment Results

During our assessment on the scoped Blackhaven core contracts, we discovered 11 findings. No critical issues were found. Three findings were of medium impact, two were of low impact, and the remaining findings were informational in nature.

6.1. Observations and recommendations

The Blackhaven codebase demonstrates solid implementation with clean architecture and appropriate use of established patterns (OpenZeppelin contracts, Uniswap V3 integration). The modular design separating concerns across Bond, BAM, Minter, and BackingCalculator is well-structured, and documentation is thorough.

The issues identified are addressable without major architectural changes. Below are key observations and recommendations for strengthening the codebase.

Inconsistent decimal handling across the protocol

Some calculations use hardcoded `1e18` while others call `.decimals()`. Given that NAV/mNAV calculations, bond pricing, and premium capture all depend on consistent decimal handling, this inconsistency could lead to incorrect calculations if backing tokens with non-18 decimals are ever introduced. We recommend standardizing the approach across all contracts, preferring dynamic `.decimals()` calls where feasible.

Liquidity operations have MEV exposure

BAM's `collectPremium` is publicly callable with configurable slippage tolerance, exposing them to sandwich attacks (see Finding 3.5. ↗). While swap sizes are constrained by `maxSupplyPercentBps` and slippage parameters are adjustable, these functions present opportunities for MEV extraction. Additionally, premium collection may fail if Uniswap liquidity is insufficient within slippage bounds, and LiquidityManager lacks the `onERC721Received` callback for safe NFT transfers.

There are no emergency mechanisms

The contracts lack pause functionality. In the event of a vulnerability or market anomaly, there is no way to halt operations without deploying new contracts. Bond, BAM, and RBTNote would benefit from pausable functionality to protect user funds during emergencies. We recommend implementing OpenZeppelin's `Pausable` pattern for these contracts.

Test coverage could expand to edge cases

The test suite covers core functionality but could be strengthened with additional coverage. End-to-end tests covering cross-contract flows — such as the full premium capture life cycle and RBTNote deposit-to-withdrawal paths — would increase confidence in system behavior. Additionally, adversarial tests targeting precision boundaries, extreme parameter values, and

failure scenarios would help catch edge cases before deployment.

6.2. 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.