Zellic

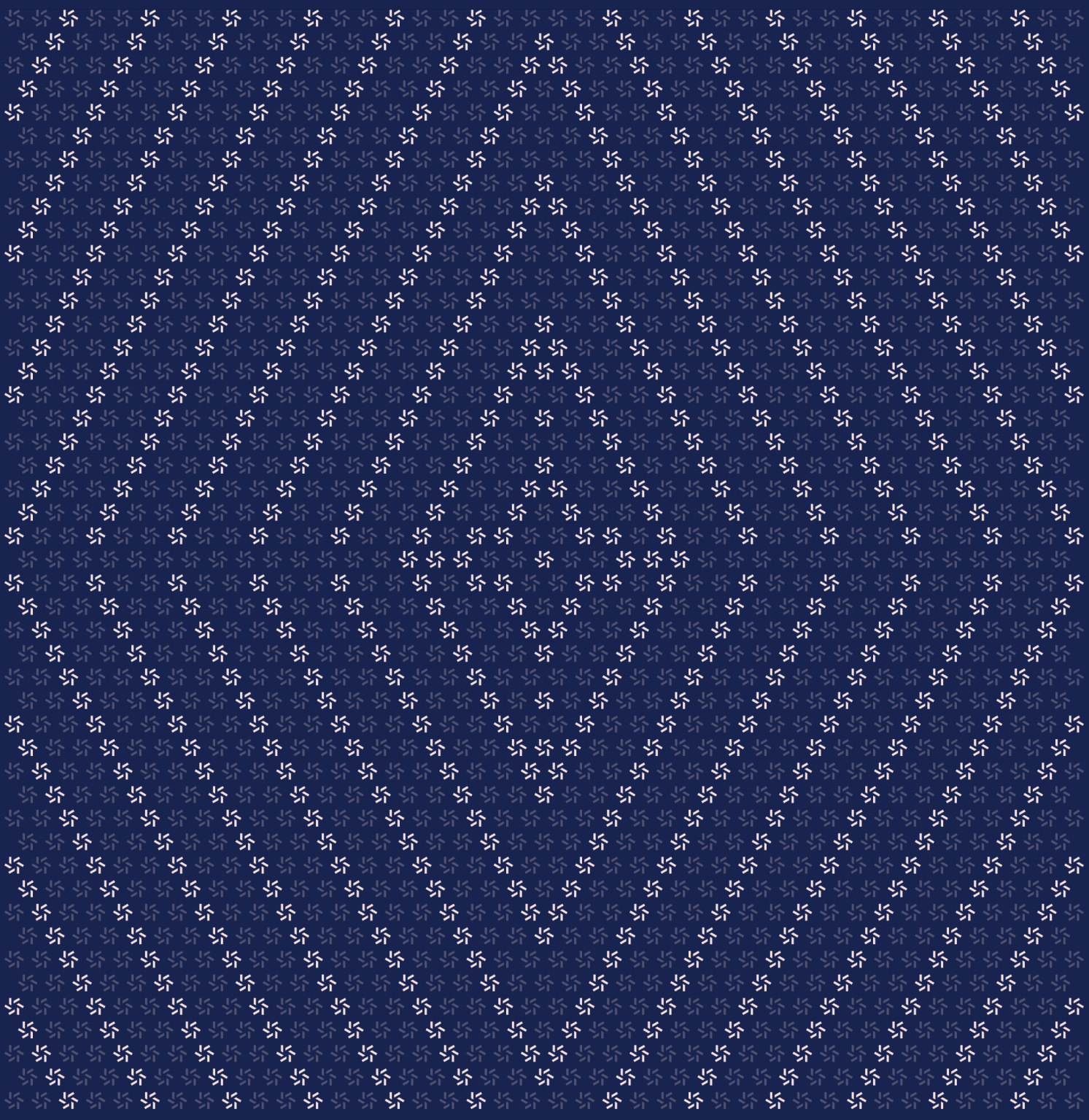
Prepared for
Dex CEO
DexFi

Prepared by
Weipeng Lai
Varun Verma
Zellic

May 5, 2025

DexFi Factory

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 DexFi from April 28th to April 29th, 2025. During this engagement, Zellic reviewed DexFi Factory'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:

- Does the factory guarantee that only whitelisted farm and profit-token beacons can ever be linked to a vault, even after future upgrades?
- Can a compromised or malicious owner/operator abuse beacon upgrades or config functions to seize or freeze user funds?
- Are fee caps, depositor limits, and pause/blacklist controls enforced strongly enough?
- Do any external calls during vault creation, whitelist updates, or harvesting expose reentrancy or delegate-call vulnerability vectors?

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.

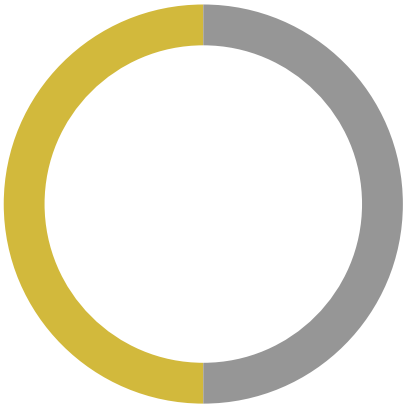
1.4. Results

During our assessment on the scoped DexFi Factory contracts, we discovered two findings. No critical issues were found. One finding was of medium impact and the other finding was informational in nature.

Additionally, Zellic recorded its notes and observations from the assessment for the benefit of DexFi in the Discussion section (4. ↗).

Breakdown of Finding Impacts

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



2. Introduction

2.1. About DexFi Factory

DexFi contributed the following description of DexFi Factory:

> DexFi offers an ecosystem of financial products designed to empower users and simplify the DeFi experience.

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:

DexFi Factory Contracts

Type	Solidity
Platform	EVM-compatible
Target	DexFi-Vaults
Repository	https://github.com/dexfinance-com/DexFi-Vaults ↗
Version	`2d0f6bd5ba35e7207b8bd98237ea8d8a795082bc`
Programs	`DexFiVaultFactory.sol`

2.4. Project Overview

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

Contact Information

The following project managers were associated with the engagement:

Jacob Goreski
Engagement Manager
jacob@zellic.io

Chad McDonald
Engagement Manager
chad@zellic.io

The following consultants were engaged to conduct the assessment:

Weipeng Lai
Engineer
weipeng.lai@zellic.io

Varun Verma
Engineer
varun@zellic.io

2.5. Project Timeline

The key dates of the engagement are detailed below.

April 28, 2025	Start of primary review period
April 29, 2025	End of primary review period

3. Detailed Findings

3.1. Lack of a whitelist check

Target	DexFiVaultFactory		
Category	Business Logic	**Severity**	High
Likelihood	Low	**Impact**	Medium

Description

The `createVault()` accepts an array of farm descriptors (`IDexFiVault.Farm[] farms_`) but never verifies that each farm is present in `_farmsWhitelist`. Although `DexFiVault.initialize()` eventually rechecks the list via `DexFiVaultHelper.processFarm`, the factory still 1) exposes users to a future implementation upgrade where the vault-side check might be removed and 2) deploys a `BeaconProxy` (wasting gas) before the revert occurs.

Impact

An attacker can supply a malicious farm contract that gains control over the vault's funds the moment any downstream developer drops or weakens the vault-side validation.

Recommendations

Add explicit allowlist validation in `createVault()` before the proxy is deployed:

```
for (uint256 i; i < farms_.length; ++i) {
    require(
        _farmsWhitelist.contains(farms_[i].beacon),
        "FARM_NOT_WHITELISTED"
    );
}
```

This keeps the factory itself the single source of truth, saves users deployment gas on invalid inputs, and prevents bugs introduced by future vault upgrades.

Remediation

This issue has been acknowledged by DexFi, and a fix was implemented in commit abb42b53 ↗.

3.2. Connector data is not cleared when profit tokens or farms are removed from the allowlist

Target	DexFiVaultFactory		
Category	Business Logic	**Severity**	Informational
Likelihood	N/A	**Impact**	Informational

Description

The `removeProfitTokensWhitelist` function does not clear data in the `profitTokenConnector` storage mapping. Consequently, calls to `profitTokenConnector` may return outdated connector information after the associated profit-token beacon is removed from the whitelist.

Similarly, the `removeFarmsWhitelist` function does not clear data in the `farmCalculationConnector` mapping.

Impact

Data queried from `profitTokenConnector` and `farmCalculationConnector` might correspond to profit tokens or farms that are no longer on the whitelist.

Recommendations

We recommend clearing the relevant connector data within the `removeProfitTokensWhitelist` and `removeFarmsWhitelist` functions when items are removed.

Remediation

This issue has been acknowledged by DexFi.

DexFi provided the following response to this finding:

> We do not remove `calculateConnector` when removing `profitToken` from the factory, as there are vaults that used this profit token when creating the vault before its removal; therefore, there is a need to access the data on this `calculateConnector`. The `removeProfitTokensWhitelist` will simply prevent the creation of new vaults with this `profitToken`.
>
> Similarly to `ProfitToken` (for `removeFarmsWhitelist`).

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. Whitelist removal is irreversible

Once a beacon is removed from the `farms` or `profitToken` whitelist, the owner cannot add the same beacon back. The `addFarmsWhitelist` and `addProfitTokensWhitelist` functions can only create new `UpgradeableBeacon`s and add them to the whitelist, resulting in a new address instead of the original beacon address. Consider allowing the direct addition of a beacon address to the whitelist in `addFarmsWhitelist` and `addProfitTokensWhitelist`.

This issue has been acknowledged by DexFi. DexFi provided the following response to this issue:

> After removing `ProfitToken` or `Farm`, it can be added as a new profit token with the same `initializeData`.

4.2. Ambiguity in the name `vaultImplementation`

The term `vaultImplementation` refers to both the beacon and the beacon's source implementation, creating ambiguity. This confusion can lead to operational errors if the owner mistakes the beacon address stored in `_vaultConfig.vaultImplementation` for the vault's source-implementation contract address. For instance, when updating fields in the `VaultConfig _vaultConfig` storage, other than `_vaultConfig.vaultImplementation`, the owner might

- read the current `VaultConfig` struct using `factory.vaultConfig()`
- modify fields in the `VaultConfig` instance but leave `_vaultConfig.vaultImplementation` unchanged, so it still contains the UpgradeableBeacon address
- call `factory.updateVaultConfig()` with the modified `VaultConfig` struct

Inside the `updateVaultConfig` function, the line `UpgradeableBeacon(previousVaultImplementation).upgradeTo(config.vaultImplementation)` instructs the UpgradeableBeacon contract to upgrade its implementation address to its own address. This action would disrupt the functionality of all vaults that depend on this beacon.

Consider using distinct struct definitions for storage and function parameters to prevent this confusion. For example, define a `VaultConfig` struct for storage and a `VaultConfigParams` struct as input for the `updateVaultConfig()` function. In the `VaultConfig` storage struct, rename the field to `vaultBeacon`. Keep the name `vaultImplementation` in the `VaultConfigParams` input struct to clearly indicate it expects the implementation address.

This issue has been acknowledged by DexFi. DexFi provided the following response to this issue:

> In case the owner confuses and sets an invalid address for `vaultImplementation`, this will only result in the inability of the vaults to operate, which can be fixed by replacing `vaultImplementation` with the correct implementation.

4.3. Misaligned validation for `initialPriceNativeAmount`

The variable `initialPriceNativeAmount` serves as the denominator in a ratio calculation involving `divider`.

```
mintAmount = totalSupply() > 0 && preDepositTotalFarmsNativeInvestments > 0
    ? (amount * totalSupply()) / preDepositTotalFarmsNativeInvestments
    : (amount * divider) / factory.vaultConfig().initialPriceNativeAmount;
```

However, the `_updateVaultConfig` function validates `initialPriceNativeAmount` by comparing it to `MIN_VAULT_MANAGEMENT_AMOUNT`, a constant representing an absolute value threshold.

```
function _updateVaultConfig(VaultConfig memory config) private {
    // [...]
    if (config.initialPriceNativeAmount < MIN_VAULT_MANAGEMENT_AMOUNT) {
        revert UpdateVaultConfigVaultInitialPriceNativeAmountUnderflow(
            config.initialPriceNativeAmount,
            MIN_VAULT_MANAGEMENT_AMOUNT
        );
    }
    // [...]
}
```

This validation does not align with the variable's intended use as part of a ratio.

Consider introducing a dedicated `MIN_INITIAL_PRICE` variable and validating `initialPriceNativeAmount` against it.

This issue has been acknowledged by DexFi. DexFi provided the following response to this issue:

> Initially, when designing the contract, several constant variables were established, and when selecting the correct values for `initialPriceNativeAmount` and `minSyntheticTransferAmount`, they were the same, so we eliminated one constant.

4.4. Test suite

The suite is a deep integration harness, not a focused unit spec for the factory. It uses mainnet forks to exercise full deposit/withdraw/harvest flows. That proves the happy path of `addFarmsWhitelist`, `updateVaultConfig`, and `createVault` works end to end, but it never stresses the factory's guardrails. Tests for the following are missing:

1. Rejecting a farm that is not whitelisted
2. Refusing a profit token that is not whitelisted

We recommend adding tests that stress the factory in isolation.

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: DexFiVaultFactory

Description

What is the component responsible for?

Deploying new `DexFiVault` instances, maintaining global fee/limit parameters, and curating the allowlists (beacon addresses) for farms and profit tokens used by every vault.

What does it do?

- Holds configuration structs `FeeConfig`, `VaultConfig`, `ProfitConfig`, and `IntegrationConfig`
- Adds/updates/removes farm and profit-token beacons
- Mints vault proxies via `createVault()`, then calls their initializer
- Exposes pause/unpause and autoharvest blacklist toggles
- Owns all beacons, allowing the owner to upgrade vault/farm/profit implementations

How does it do it?

- It uses OpenZeppelin `UpgradeableBeacon` plus `BeaconProxy`.
- The `createVault()` spins up a `BeaconProxy` that references the global vault beacon and forwards the user-supplied params.
- The `OnlyOwner` functions mutate the config structs and upgrade beacons when needed.

Invariants

- Only the owner or operator can mutate allowlists — `onlyOwner` / `onlyOwnerOrOperator` modifiers.
- Farms / profit tokens used by a vault must be whitelisted. This is implicitly checked inside `DexFiVaultHelper.processFarm`, *not* in `createVault()`.
- Vault and profit beacons are never zero address.

Test coverage

Cases covered

- Happy-path flow — whitelist a beacon -> update vault beacon -> `createVault()` -> deposit/harvest/withdraw
- Updating farms via `updateFarmsWhitelist()`

Cases not covered

- Reverting when supplying a farm beacon that is *not* whitelisted
- Reverting when supplying an unwhitelisted profit token (with `profit_ > 0`)
- Auth tests — only owner/operator can call whitelist mutators, and only owner can change configs

Attack surface

- **Functions**
  - `createVault()` (anyone) — an attacker controls `name`, `symbol`, `profit_`, `profitToken_`, and the full `farms_` array
  - `addFarmsWhitelist()` / `addProfitTokensWhitelist()` (owner/operator) — relies on external `initialize()` calls from untrusted farm/profit contracts
  - `update…Config()` (owner) — can brick or drain the ecosystem if caps are ignored
  - `toggleVaultAutoHarvest()` (vault owners) — address param not checked against `_vaults`, enabling blacklist grief
- **Upgradable beacons.** Factory owner can push arbitrary code to every vault, farm, and profit token in a single TX.
- **Implicit trust in vault implementation.** All farm allowlist checks currently sit inside the vault; if the beacon is upgraded to a version without those checks, the factory provides no fallback guard. We recommend to add explicit whitelist validation in `createVault()`; see Finding 3.1. ↗.

6. Assessment Results

During our assessment on the scoped DexFi Factory contracts, we discovered two findings. No critical issues were found. One finding was of medium impact and the other finding was informational in nature.

The repository appears functional, and the integration tests confirm that deposits, harvests, whitelist management, and basic upgrades work when all inputs are valid. Coverage is still thin on adversarial scenarios, because there are few tests that try invalid beacons, boundary fee values, unauthorized callers, or other edge inputs that should trigger reverts. Adding fuzz campaigns in Foundry, together with explicit unit tests for every require branch, would give clearer evidence that the factory and vaults behave safely under stress.

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.