Zellic

Sturdy

Smart Contract Security Assessment

September 15, 2023

Prepared for:

Sam Forman

Sturdy

Prepared by:

Sina Pilehchiha and Yuhang Wu

Zellic Inc.

Contents

About Zellic

Zellic was founded in 2020 by a team of blockchain specialists with more than a decade of combined industry experience. We are leading experts in smart contracts and Web3 development, cryptography, web security, and reverse engineering. Before Zellic, we founded perfect blue, the top competitive hacking team in the world. Since then, our team has won countless cybersecurity contests and blockchain security events.

Zellic aims to treat clients on a case-by-case basis and to consider their individual, unique concerns and business needs. Our goal is to see the long-term success of our partners rather than simply provide a list of present security issues. Similarly, we strive to adapt to our partners' timelines and to be as available as possible. To keep up with our latest endeavors and research, check out our website zellic.io or follow @zellic_io on Twitter. If you are interested in partnering with Zellic, please contact us at hello@zellic.io.

1 Executive Summary

Zellic conducted a security assessment for Sturdy from September 11th to September 13th, 2023. During this engagement, Zellic reviewed Sturdy's code for security vulnerabilities, design issues, and general weaknesses in security posture.

1.1 Goals of the Assessment

In a security assessment, goals are framed in terms of questions that we wish to answer. These questions are agreed upon through close communication between Zellic and the client. In this assessment, we sought to answer the following questions:

- Could an on-chain attacker or a malicious user trigger a loss of user funds?
- Is the business logic of manual allocation performing as expected?
- Is the `requestLiquidity()` function providing just-in-time liquidity as expected?

1.2 Non-goals and Limitations

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

- Lender, oracle, and swapper modules of the codebase
- The Yearn V3 Vault that in-scope contracts interact with
- 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.

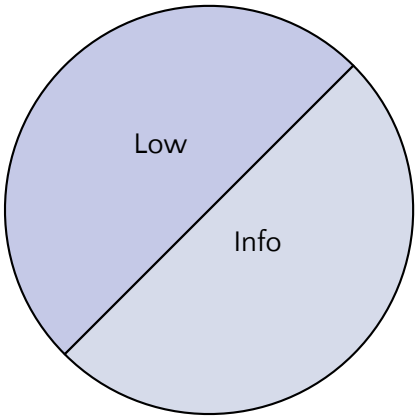
1.3 Results

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

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

Breakdown of Finding Impacts

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



2 Introduction

2.1 About Sturdy

Sturdy enables anyone to create a liquid money market for any token. Sturdy uses a novel two-tier architecture to isolate risk between assets while avoiding liquidity fragmentation. The base layer consists of risk-isolated pools; aggregation built on top enables lenders to select which collateral assets can be used as collateral for their deposits.

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, upgradeability weaknesses, centralization risks, and so on.

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

Zellic organizes its reports such that the most important findings come first in the document, rather than being strictly ordered on impact alone. Thus, we may sometimes emphasize an "Informational" finding higher than a "Low" finding. The key distinction is that although certain findings may have the same impact rating, their *importance* may differ. This varies based on various soft factors, like our clients' threat models, their business needs, and so on. We aim to provide useful and actionable advice to our partners considering their long-term goals, rather than a simple list of security issues at present.

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:

Sturdy Contracts

Repository	https://github.com/sturdyfi/sturdy-aggregator-new/tree/dev
Version	sturdy-aggregator-new: `6ee0b32613ae239c349399690392d56ab24faa7b`
Programs	DebtManager SiloGateway
Type	Solidity
Platform	EVM-compatible

2.4 Project Overview

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

Sina Pilehchiha, Engineer
sina@zellic.io

Yuhang Wu, Engineer
yuhang@zellic.io

2.5 Project Timeline

The key dates of the engagement are detailed below.

September 11, 2023	Kick-off call
September 11, 2023	Start of primary review period
September 13, 2023	End of primary review period

3 Detailed Findings

3.1 Unhandled division-by-zero error in `borrowAsset()`

- **Target**: SiloGateway
- **Category**: Coding Mistakes
- **Likelihood**: N/A
- **Severity**: Informational
- **Impact**: Informational

Description

In the `borrowAsset()` function, there is no check for the possibility of `totalAsset` being 0, which could lead to a division-by-zero error in `numerator / totalAsset`.

```
function borrowAsset(
        address _silo,
        uint256 _borrowAmount,
        uint256 _collateralAmount,
        address _collateralAsset,
        address _receiver
    ) external nonReentrant {
        (uint256 totalAsset, ) = ISilo(_silo).totalAsset();
        (uint256 totalBorrow, ) = ISilo(_silo).totalBorrow();
        uint256 numerator = UTIL_PREC * (totalBorrow + _borrowAmount);
        uint256 utilizationRate = numerator / totalAsset;
        ...
```

Impact

If `totalAsset` is 0 and someone tries to execute the `borrowAsset`, the transaction will revert due to the division by zero.

Recommendations

Add a zero check on `totalAsset` for a more graceful and informative handling of this situation.

```
require(totalAsset ≠ 0, "Total Asset is zero");
```

Remediation

This issue has been acknowledged by Sturdy, and a fix was implemented in commit ca396917.

3.2 Missing test suite code coverage

- **Target**: DebtManager, SiloGateway
- **Category**: Code Maturity
- **Likelihood**: Low
- **Severity**: Low
- **Impact**: Low

Description

Some functions in the smart contract are not covered by any unit or integration tests, to the best of our knowledge. The following functions do not have full test coverage:

DebtManager.sol: `zkAllocation`, `sortLendersWithAPR`, `setZKVerifier`, `setWhitelistedGateway`, `setPairToLender`, `setAprOracle`, `requestLiquidity`, and `manualAllocation`.

SiloGateway.sol: `setUtilizationLimit` and `borrowAsset`.

Impact

Because correctness is so critically important when developing smart contracts, we recommend that all projects strive for 100% code coverage. Testing should be an essential part of the software development life cycle. No matter how simple a function may be, untested code is always prone to bugs.

Recommendations

Expand the test suite so that all functions and their branches are covered by unit or integration tests.

Remediation

This issue has been acknowledged by Sturdy, and a fix was implemented in commit dc16e5a8.

This finding reflects the project's test coverage at commit 6ee0b326. During the remediation process Sturdy implemented additional test coverage as outlined in Section 5.

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 Inefficient Loop Operations in DebtManager

In DebtManager, there are several loop operations. The reason is that the `array` structure is used to store `lenders` to facilitate sorting. However, in practice, if the `lenders` array becomes very large, the overhead of many functions in the entire contract will increase significantly.

Currently, the sorting method used is bubble sort. This is more gas efficient for small arrays, but it consumes more gas for larger arrays. Given that the `lenders` are used in multiple functions and have a significant impact, it is recommended to use a `mapping` structure to store `lenders`.

For sorting, according to the best implementation, one possible suggestion is to perform it off chain. In this way, the overall overhead will be similar, regardless of how large the `lenders` array is.

4.2 Centralization risks

The `owner(admin)` of DebtManager can modify key configurations such as adding or removing lenders, setting the APR oracle, linking external pool addresses to lenders, and adjusting whitelisted gateways.

In the unfortunate event that the owner's account is compromised, there is a potential risk to user funds. As a precautionary measure, we recommend implementing enhanced operational security practices, such as utilizing multi-signature wallets, to bolster the security of the system and safeguard user assets.

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: DebtManager.sol

Function: `addLender(address _lender)`

This adds lender to list.

Inputs

- `_lender`
  - **Control**: Full.
  - **Constraints**: Needs to be a valid address.
  - **Impact**: The lender to manage debt.

Branches and code coverage (including function calls)

Intended branches

- Owner can successfully add lender.
  ☑ Test coverage

Negative behavior

- Function reverts if invoked by nonowner.
  ☑ Negative test
- Function reverts if invoked by owner with a strategy with its `aggregator.strategies(_lender).activation` being equal to zero.
  ☑ Negative test

Function call analysis

- `rootFunction` → `aggregator.strategies(address)`
  - **What is controllable?** `_lender`.

  - **If return value controllable, how is it used and how can it go wrong?** It is used to determine whether the strategy for `_lender` is revoked or not.
  - **What happens if it reverts, reenters, or does other unusual control flow?** N/A.

Function: `manualAllocation(LenderAllocation[] _newPositions)`

This manually updates the allocations.

Inputs

- `_newPositions`
  - **Control**: Full.
  - **Constraints**: Should be in order of decreasing debt and increasing debt.
  - **Impact**: The list of position info.

Branches and code coverage (including function calls)

Intended branches

- Owner successfully updates the allocations.
  ☑ Test coverage

Negative behavior

- Function reverts if invoked by nonowner.
  ☑ Negative test
- Function reverts because `position.debt` is greater than `lenderData.max_debt` (supply cap).
  ☑ Negative test
- Function reverts because `lenderData.activation` for one of the strategies is unactivated (in `_manualAllocation()`).
  ☑ Negative test

Function call analysis

- `rootFunction` → `_manualAllocation(LenderAllocation[])`
  - **What is controllable?** `_newPositions`.
  - **If return value controllable, how is it used and how can it go wrong?** N/A.
  - **What happens if it reverts, reenters, or does other unusual control flow?** N/A.

Function: `removeLender(address _lender)`

This removes lender from list.

Inputs

- `_lender`
  - **Control**: Full.
  - **Constraints**: Needs to be a valid address.
  - **Impact**: The lender to manage debt.

Branches and code coverage (including function calls)

Intended branches

- Owner can successfully remove lender.
  ☑ Test coverage
- Owner or nonowner can successfully remove lender with a strategy with its `aggregator.strategies(_lender).activation` being equal to zero.
  ☑ Test coverage

Negative behavior

- Function reverts if invoked by nonowner.
  ☑ Negative test

Function call analysis

- `rootFunction` → `aggregator.strategies(address)`
  - **What is controllable?** `_lender`
  - **If return value controllable, how is it used and how can it go wrong?** It is used to determine whether the strategy for `_lender` is revoked or not.
  - **What happens if it reverts, reenters, or does other unusual control flow?** N/A.

Function: `requestLiquidity(uint256 _amount, address _pair)`

This processes the just-in-time liquidity.

Inputs

- `_amount`
  - **Control**: Full.
  - **Constraints**: N/A.

  - **Impact**: The required liquidity amount.
- `_pair`
  - **Control**: Full.
  - **Constraints**: Only whitelisted gateways can request liquidity in case of borrow.
  - **Impact**: The silo address.

Branches and code coverage (including function calls)

Intended branches

- User successfully triggers `requestLiquidity` for just in-time liquidity.
  ☑ Test coverage

Negative behavior

- Function reverts if caller is not whitelisted.
  ☑ Negative test
- FUnction reverts if there is no mapping between silo and strategy
  ☑ Negative test
- Function reverts if the sum of `requestingLenderData.current_debt` and `_amount` is not less than `requestingLenderData.max_debt - 1`.
  ☑ Negative test
- Function reverts if the sum of `requestingLenderData.current_debt` and `_amount` is not less than the total assets of the aggregator (`aggregator.totalAssets()`).
  ☑ Negative test
- Function reverts if requiredAmount is not equal to zero at the end of the process of withdrawal from other lenders to fill the required amount.
  ☑ Negative test

Function call analysis

- `rootFunction → aggregator.strategies(address)`
  - **What is controllable?** N/A.
  - **If return value controllable, how is it used and how can it go wrong?** N/A.
  - **What happens if it reverts, reenters, or does other unusual control flow?** N/A.
- `rootFunction → aggregator.totalAssets()`
  - **What is controllable?** N/A.
  - **If return value controllable, how is it used and how can it go wrong?** N/A.
  - **What happens if it reverts, reenters, or does other unusual control flow?** N/A.

- `rootFunction → aggregator.totalIdle()`
  - **What is controllable?** N/A.
  - **If return value controllable, how is it used and how can it go wrong?** N/A.
  - **What happens if it reverts, reenters, or does other unusual control flow?** N/A.
- `rootFunction → aggregator.minimum_total_idle()`
  - **What is controllable?** N/A.
  - **If return value controllable, how is it used and how can it go wrong?** N/A.
  - **What happens if it reverts, reenters, or does other unusual control flow?** N/A.
- `rootFunction → aggregator.assess_share_of_unrealised_losses(address, uint256)`
  - **What is controllable?** N/A.
  - **If return value controllable, how is it used and how can it go wrong?** N/A.
  - **What happens if it reverts, reenters, or does other unusual control flow?** N/A.
- `rootFunction → aggregator.process_report(address)`
  - **What is controllable?** N/A.
  - **If return value controllable, how is it used and how can it go wrong?** N/A.
  - **What happens if it reverts, reenters, or does other unusual control flow?** N/A.
- `rootFunction → aggregator.update_debt(address, uint256)`
  - **What is controllable?** `_amount` in `requestingLenderData.current_debt + _amount`, which is the `target_debt`.
  - **If return value controllable, how is it used and how can it go wrong?** N/A.
  - **What happens if it reverts, reenters, or does other unusual control flow?** N/A.

Function: `setAprOracle(IAprOracle _aprOracle)`

This sets the APR oracle contract address.

Inputs

- `_aprOracle`
  - **Control**: Full.
  - **Constraints**: Needs to be a valid address.
  - **Impact**: The oracle contract address.

Branches and code coverage (including function calls)

Intended branches

- Owner successfully invokes function.
  ☑ Test coverage

Negative behavior

- Function reverts if invoked by nonowner.
  ☑ Negative test

Function: `setPairToLender(address _pair, address _lender)`

This sets the lender's external pool address.

Inputs

- `_pair`
  - **Control**: Full.
  - **Constraints**: Needs to be a valid address.
  - **Impact**: The external pool address.
- `_lender`
  - **Control**: Full.
  - **Constraints**: Needs to be a valid address.
  - **Impact**: The lender address to manage debt.

Branches and code coverage (including function calls)

Intended branches

- Admin successfully sets the lender's external pool address.
  ☑ Test coverage

Negative behavior

- Function reverts if invoked by non-admin.
  ☑ Negative test

Function: `setWhitelistedGateway(address _gateway, bool _enabled)`

This sets the whitelisted gateway.

Inputs

- `_gateway`
  - **Control**: Full.
  - **Constraints**: Needs to be a valid address.
  - **Impact**: The Silo Gateway address.
- `_enabled`
  - **Control**: Full.
  - **Constraints**: N/A.
  - **Impact**: True if whitelisted gateway — else false.

Branches and code coverage (including function calls)

Intended branches

- Admin successfully sets a whitelisted gateway to `True`.
  ☑ Test coverage
- Admin successfully sets a whitelisted gateway to `False`.
  ☑ Test coverage

Negative behavior

- Function reverts if invoked by nonowner.
  ☑ Negative test

Function: `setZKVerifier(address _verifier)`

This sets the zero-knowledge verifier address.

Inputs

- `_verifier`
  - **Control**: Full.
  - **Constraints**: Needs to be a valid address.
  - **Impact**: The zero-knowledge verifier address.

Branches and code coverage (including function calls)

Intended branches

- Owner successfully invokes function.
  ☑ Test coverage

Negative behavior

- Function reverts if invoked by nonowner.
  ☑ Negative test

Function: `sortLendersWithAPR()`

This sorts the registered lenders based on the APR value and cache.

Branches and code coverage (including function calls)

Intended branches

- Owner successfully sorts the lenders.
  ☑ Test coverage

Negative behavior

- Function reverts if invoked by nonowner.
  ☑ Negative test

Function call analysis

- `rootFunction → getExpectedApr(address, int256)`
  - **What is controllable?** N/A.
  - **If return value controllable, how is it used and how can it go wrong?** N/A.
  - **What happens if it reverts, reenters, or does other unusual control flow?** Will revert if an oracle is not set.

Function: `zkAllocation(LenderAllocation[] _newPositions)`

This manually updates the allocations from the ZK verifier.

Inputs

- `_newPositions`
  - **Control**: Full.
  - **Constraints**: Should be in order of decreasing debt and increasing debt AND must match with `_lenders` in length (`_lenders.length == _newPositions.length`).
  - **Impact**: The list of position info.

Branches and code coverage (including function calls)

Intended branches

- Owner successfully updates the allocations from the ZK verifier.
  ☑ Test coverage

Negative behavior

- Function reverts if not invoked by `_zkVerifier`.
  ☑ Negative test
- Function reverts because `position.debt` is greater than `lenderData.max_debt` (supply cap).
  ☑ Negative test
- Function reverts because `lenderData.activation` for one of the strategies is unactivated (in `_manualAllocation()`).
  ☑ Negative test

Function call analysis

- `rootFunction` → `_manualAllocation(LenderAllocation[])`
  - **What is controllable?** `_newPositions`.
  - **If return value controllable, how is it used and how can it go wrong?** N/A.
  - **What happens if it reverts, reenters, or does other unusual control flow?** N/A.

5.2 Module: SiloGateway.sol

Function: `borrowAsset(address _silo, uint256 _borrowAmount, uint256 _collateralAmount, address _collateralAsset, address _receiver)`

This borrows asset from `_silo`.

Inputs

- `_silo`
  - **Control**: Full.
  - **Constraints**: Needs to be a valid `_silo` address.
  - **Impact**: The `_silo` address.
- `_borrowAmount`
  - **Control**: Full.
  - **Constraints**: N/A.
  - **Impact**: The borrowing amount.
- `_collateralAmount`
  - **Control**: Full.

  - **Constraints**: N/A (delegated to silo).
  - **Impact**: The collateral amount.
- `_collateralAsset`
  - **Control**: Full.
  - **Constraints**: Needs to be the same as the silo's asset (not enforced by gateway).
  - **Impact**: The collateral asset address.
- `_receiver`
  - **Control**: Full.
  - **Constraints**: Needs to be a valid address.
  - **Impact**: The receiver address of borrowing asset.

Branches and code coverage (including function calls)

Intended branches

- User successfully borrows asset from `_silo` with `_collateralAmount` being greater than zero.
  ☑ Test coverage
- User successfully borrows asset from `_silo` with `_collateralAmount` being equal to zero.
  ☑ Test coverage
- User successfully borrows asset from `_silo` with `_borrowAmount` being equal to zero.
  ☑ Test coverage

Negative behavior

- Function reverts when `_silo` is not a valid address.
  ☑ Negative test
- Function reverts when `_collateralAsset` is not the same as the silo's asset.
  ☑ Negative test
- Function reverts when `_receiver` is not a valid address.
  ☑ Negative test
- Function reverts when `totalAsset` amounts to zero.
  ☑ Negative test

Function call analysis

- `rootFunction → ISilo(address).totalAsset()`
  - **What is controllable?** N/A.
  - **If return value controllable, how is it used and how can it go wrong?** It can

go wrong by amounting to be zero.
  - **What happens if it reverts, reenters, or does other unusual control flow?** N/A.
- `rootFunction → ISilo(address).totalBorrow()`
  - **What is controllable?** N/A.
  - **If return value controllable, how is it used and how can it go wrong?** N/A.
  - **What happens if it reverts, reenters, or does other unusual control flow?** N/A.
- `rootFunction → ISilo(address).asset()`
  - **What is controllable?** N/A.
  - **If return value controllable, how is it used and how can it go wrong?** It can go wrong by not being equal to `address(asset)`.
  - **What happens if it reverts, reenters, or does other unusual control flow?** N/A.
- `rootFunction → manager.getLenderFromPair(address)`
  - **What is controllable?** `_silo` address.
  - **If return value controllable, how is it used and how can it go wrong?** It can go wrong by being equal to `address(0)`.
  - **What happens if it reverts, reenters, or does other unusual control flow?**: N/A.
- `rootFunction → manager.requestLiquidity(uint256, address)`
  - **What is controllable?** `_silo` address.
  - **If return value controllable, how is it used and how can it go wrong?** The entire `borrowAsset` transaction will revert.
  - **What happens if it reverts, reenters, or does other unusual control flow?** N/A.
- `rootFunction → ISilo(address).borrowAsset(uint256, uint256, address)`
  - **What is controllable?** `_silo` address, `_borrowAmount`, `_collateralAmount`, and `_receiver`.
  - **If return value controllable, how is it used and how can it go wrong?** It can go wrong if `_collateralAsset` is not the same as the silo's asset.
  - **What happens if it reverts, reenters, or does other unusual control flow?** The entire `borrowAsset` transaction will revert.

Function: `setUtilizationLimit(uint256 utilizationLimit_)`

This sets the utilization-limit value for the just-in-time liquidity.

Inputs

- `utilizationLimit_`
  - **Control**: Full.
  - **Constraints**: Needs to be between 0% and 100%.
  - **Impact**: The utilization limit value — 1% = 1,000.

Branches and code coverage (including function calls)

Intended branches

- Owner successfully sets `utilizationLimit_`.
  ☑ Test coverage

Negative behavior

- Function reverts due to an invalid value set for `utilizationLimit_`.
  ☑ Negative test
- Function reverts due to the caller not being the owner.
  ☑ Negative test

6 Assessment Results

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

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