Zellic

**Prepared for**
**Leonardo Saturnino**
Threshold Network

**Prepared by**
**Junghoon Cho**
**Sunwoo Hwang**
Zellic

**April 23, 2025**

# tBTC Sui Integration

## Sui Application 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 Threshold Network from April 11th to April 15th, 2025. During this engagement, Zellic reviewed tBTC Sui Integration'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:

- Can an attacker mint/burn an arbitrary amount of tBTC?
- Can an attacker bridge an arbitrary amount of tBTC?
- Can an attacker steal an arbitrary amount of bridge tokens?

## 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

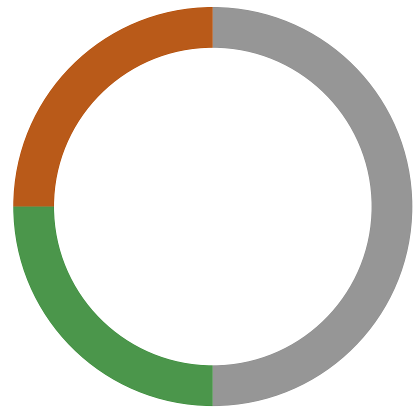
Due to the time-boxed nature of security assessments in general, there are limitations in the coverage an assessment can provide.

## 1.4. Results

During our assessment on the scoped tBTC Sui Integration contracts, we discovered four findings. No critical issues were found. One finding was of high impact, one was of low impact, and the remaining findings were informational in nature.

## Breakdown of Finding Impacts

| Impact Level | Count |
|---|---|
| Critical | 0 |
| High | 1 |
| Medium | 0 |
| Low | 1 |
| Informational | 2 |



# 2. Introduction

## 2.1. About tBTC Sui Integration

Threshold Network contributed the following description of tBTC Sui Integration:

> Threshold Network powers tBTC, the Bitcoin standard for DeFi — a trust-minimized way to bring BTC liquidity into the crypto ecosystem while always maintaining a direct settlement path back to native Bitcoin. Unlike other BTC derivatives or wrapped tokens, tBTC is a decentralized, permissionless Bitcoin bridge designed to bring Bitcoin (BTC) liquidity to Ethereum and other EVM and NON-EVM networks. It allows Bitcoin holders to mint tBTC, an ERC-20 token fully backed 1:1 by BTC, enabling seamless participation in Ethereum's DeFi ecosystem without relying on centralized intermediaries.

## 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.

## 2.3. Scope

The engagement involved a review of the following targets:

### tBTC Sui Integration Contracts

| | |
|---|---|
| **Type** | move |
| **Platform** | Sui |

| | |
|---|---|
| **Target** | tbtc-sui-integration |
| **Repository** | https://github.com/threshold-network/tbtc-sui-integration ↗ |
| **Version** | `20a094d939663a8d51ec9233d49a7b5235ec4cf7` |
| **Programs** | `bitcoin_depositor/bitcoin_depositor.move`<br>`gateway/helpers.move`<br>`gateway/wormhole_gateway.move`<br>`token/tbtc.move` |

## 2.4. Project Overview

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

**Junghoon Cho**
Engineer
junghoon@zellic.io ↗

**Sunwoo Hwang**
Engineer
sunwoo@zellic.io ↗

## 2.5. Project Timeline

The key dates of the engagement are detailed below.

| | |
|---|---|
| **April 11, 2025** | Kick-off call |
| **April 11, 2025** | Start of primary review period |
| **April 15, 2025** | End of primary review period |

## 3. Detailed Findings

### 3.1. Incorrect decimal handling

| Target | wormhole_gateway.move | | |
|---|---|---|---|
| Category | Coding Mistakes | Severity | High |
| Likelihood | High | Impact | High |

#### Description

The tBTC token in Sui has been configured with nine decimals, while its wrapped token in Wormhole supports a maximum of eight decimals.

```
/// Module initializer
fun init(witness: TBTC, ctx: &mut TxContext) {
    let (treasury_cap, metadata) = coin::create_currency(
        witness,
        9, // Bitcoin uses 8 decimals, but many chains use 9 for tBTC
        b"TBTC",
        b"Threshold Bitcoin",
        b"Canonical L2/sidechain token implementation for tBTC",
    // [...]
```

When bridging tBTC tokens to Sui, the amount should be converted from nine decimals to eight decimals. However, the current minting function incorrectly uses the same amount for the wrapped token as for the tBTC tokens, without performing the necessary decimal conversion.

```
// Redeem the coins
let (
    bridged_coins,
    _parsed_transfer,
    _source_chain,
) = complete_transfer_with_payload::redeem_coin(&capabilities.emitter_cap,
    receipt);

// Get the amount of tokens to mint
let amount = coin::value(&bridged_coins);
// [...]
 // Mint TBTC tokens
    TBTC::mint(
        &capabilities.minter_cap,
        &mut capabilities.treasury_cap,
        token_state,
```

```
        amount, // @audit-issue use same wrapped coin amount for TBTC token
        recipient,
        ctx,
    );
```

## Impact

Users will receive fewer tBTC tokens than expected when bridging to Sui, and similarly, they will receive fewer tBTC tokens when bridging from Sui to other chains due to this incorrect decimal handling.

## Recommendations

Use eight decimals for tBTC tokens or implement decimal conversion between tBTC and wrapped tokens.

## Remediation

This issue has been acknowledged by Threshold Network, and a fix was implemented in commit `65ecaee4` ↗.

## 3.2. Potential nonce reuse

| | | | |
|---|---|---|---|
| **Target** | wormhole_gateway.move | | |
| **Category** | Coding Mistakes | **Severity** | Low |
| **Likelihood** | Medium | **Impact** | Low |

### Description

The `send_token` and `send_wrapped_tokens` functions receive a `nonce` parameter, which is used to make the wormhole message. The nonce in wormhole message can be any user-defined value to uniquely identify the message. However, the current implementation does not check if the nonce has already been used.

### Impact

While not a security issue, allowing the reuse of nonces can lead to confusion when identifying the message.

### Recommendations

Implement a check to ensure the nonce has not been used before.

### Remediation

This issue has been acknowledged by Threshold Network, and a fix was implemented in commit `8fe74500` ↗.

## 3.3. Lack of checks for pause and unpause

| | | | |
|---|---|---|---|
| **Target** | wormhole_gateway.move | | |
| **Category** | Coding Mistakes | **Severity** | Informational |
| **Likelihood** | N/A | **Impact** | Informational |

### Description

The pause and unpause functions lack state-validation checks. These functions do not verify if the gateway is already in the target state (paused/unpaused) before executing the state change.

```
public entry fun pause(_: &AdminCap, state: &mut GatewayState, _ctx:
    &mut TxContext) {
    // Verify the gateway is initialized
    assert!(state.is_initialized, E_NOT_INITIALIZED);

    state.paused = true;
    event::emit(Paused {});
}
```

### Impact

While this is not a security issue, it can lead to misleading events being emitted when calling these functions on a gateway that is already in the target state.

### Recommendations

Add state-validation checks to the pause and unpause functions to ensure the gateway is in the appropriate state before executing the state change.

### Remediation

This issue has been acknowledged by Threshold Network, and a fix was implemented in commit 7b4c86e1 ↗.

### 3.4. Redundant checks

| | | | |
|---|---|---|---|
| **Target** | tbtc.move | | |
| **Category** | Coding Mistakes | **Severity** | Informational |
| **Likelihood** | N/A | **Impact** | Informational |

## Description

The `remove_minter` function uses the `is_minter` function to check if the address is in the minters list. This is redundant as the `vector::index_of` function also checks if the address is in the list. This is the same for the `remove_guardian` function.

```
public fun is_minter(state: &TokenState, addr: address): bool {
    vector::contains(&state.minters, &addr)
}
// [...]
public entry fun remove_minter(
    _: &AdminCap,
    state: &mut TokenState,
    minter: address,
    _ctx: &mut TxContext,
) {
    assert!(is_minter(state, minter), E_NOT_IN_MINTERS_LIST);

    let (found, index) = vector::index_of(&state.minters, &minter);
    assert!(found, E_NOT_IN_MINTERS_LIST);
    // [...]
```

## Impact

While not a security issue, these redundant checks increase unnecessary gas costs and reduce code efficiency.

## Recommendations

Remove the redundant checks.

## Remediation

This issue has been acknowledged by Threshold Network, and a fix was implemented in commit `65ecaee4` ↗.

# 4. 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 ways an attacker may approach breaking a given function.

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.

## 4.1. Contract: tbtc.move

### Function: `add_minter(_: &AdminCap, state: &mut TokenState, minter: address, ctx: &mut TxContext)`

The `add_minter` function adds an address as a minter to `TokenState` and sends `MinterCap` to that address.

- ☑ The caller must provide a valid `AdminCap`.
- ☑ It must ensure that `minter` is not registered in `state.minters` already.
- ☑ The `minter` is newly registered in `state.minters`.
- ☑ `MinterCap` is created and transferred to `minter`.

### Function: `remove_minter(_: &AdminCap, state: &mut TokenState, minter: address, _ctx: &mut TxContext)`

The `remove_minter` function removes a minter from `TokenState`.

- ☑ The caller must provide a valid `AdminCap`.
- ☑ It must ensure that `minter` is registered in `state.minters` already.
- ☑ The `minter` is removed from `state.minters`.

### Function: `add_guardian(_: &AdminCap, state: &mut TokenState, guardian: address, ctx: &mut TxContext)`

The `add_guardian` function adds an address as a guardian to `TokenState` and sends `GuardianCap` to that address.

- ☑ The caller must provide a valid `AdminCap`.
- ☑ It must ensure that `guardian` is not registered in `state.guardians` already.
- ☑ The `guardian` is newly registered in `state.guardians`.
- ☑ `GuardianCap` is created and transferred to `guardian`.

#### Function: `remove_guardian(_: &AdminCap, state: &mut TokenState, guardian: address, _ctx: &mut TxContext)`

The `remove_guardian` function removes a guardian from `TokenState`.

- ☑ The caller must provide a valid `AdminCap`.
- ☑ It must ensure that `guardian` is registered in `state.guardians`.
- ☑ The `guardian` is removed from `state.guardians`.

#### Function: `unpause(_: &AdminCap, state: &mut TokenState, ctx: &mut TxContext)`

The `unpause` function unpauses the `state`, allowing any interactions.

- ☑ The caller must provide a valid `AdminCap`.
- ☑ It must ensure that `state.paused` is `true`.
- ☑ `state.paused` is set to `false`.

#### Function: `pause(_: &GuardianCap, state: &mut TokenState, ctx: &mut TxContext)`

The `pause` function pauses the `state`, disallowing any interactions.

- ☑ The caller must provide a valid `GuardianCap`.
- ☑ It must ensure that the caller is in `state.guardians`.
- ☑ It must ensure that `state.paused` is `false`.
- ☑ `state.paused` is set to `true`.

#### Function: `mint(_: &MinterCap, treasury_cap: &mut TreasuryCap<TBTC>, state: &TokenState, amount: u64, recipient: address, ctx: &mut TxContext)`

The `mint` function mints an `amount` of tBTC and transfers it to the `recipient`.

- ☑ The caller must provide a valid `MinterCap` and `TreasuryCap<TBTC>`.
- ☑ It must ensure that the caller is in `state.minters`.
- ☑ It must ensure that `state.paused` is `false`.
- ☑ An `amount` of tBTC is minted and transferred to the `recipient`.

#### Function: `burn(treasury_cap: &mut TreasuryCap<TBTC>, state: &TokenState, coin: Coin<TBTC>)`

The `burn` function burns the supplied tBTC token.

- ☑ The caller must provide a valid `TreasuryCap<TBTC>`.
- ☑ It must ensure that `state.paused` is `false`.
- ☑ The supplied tBTC token is burned.

## 4.2. Contract: bitcoin_depositor.move

### Function: `set_trusted_emitter(_: &AdminCap, state: &mut ReceiverState, emitter: vector<u8>, _ctx: &mut TxContext)`

The `set_trusted_emitter` function updates the trusted emitter address for the `ReceiverState`.

- ☑ The caller must provide a valid `AdminCap`.
- ☑ `state.trusted_emitter` is updated to `emitter`.

### Function: `initialize_deposit(funding_tx: vector<u8>, deposit_reveal: vector<u8>, deposit_owner: vector<u8>, ctx: &mut TxContext)`

The `initialize_deposit` function initializes a new deposit with the provided details.

- ☑ No capability is required; any caller may invoke it.
- ☑ It emits a `DepositInitialized` event with deposit details including `funding_tx`, `deposit_reveal`, `deposit_owner`, and `sender`.

### Function: `receiveWormholeMessages<CoinType>(receiver_state: &mut ReceiverState, gateway_state: &mut Gateway::GatewayState, capabilities: &mut Gateway::GatewayCapabilities, treasury: &mut Gateway::WrappedTokenTreasury<CoinType>, wormhole_state: &mut WormholeState, token_bridge_state: &mut token_bridge::state::State, token_state: &mut TBTC::TokenState, vaa_bytes: vector<u8>, clock: &Clock, ctx: &mut TxContext)`

The `receiveWormholeMessages` function is called by the relayer to receive messages from Wormhole. It verifies the message and then calls `Gateway::redeem_tokens` to redeem the bridged tokens on Sui.

- ☑ It verifies the VAA's authenticity.
- ☑ It must ensure that the VAA hash has not been processed before.
- ☑ It must ensure `emitter_chain == EMITTER_CHAIN_L1`.
- ☑ It must ensure `emitter_address == state.trusted_emitter`.
- ☑ It marks the VAA hash as processed by adding it to `receiver_state.processed_vaas`.
- ☑ It emits a `MessageProcessed` event with the VAA hash.
- ☑ It calls `Gateway::redeem_tokens` to redeem the bridged tokens on Sui.

## 4.3. Contract: wormhole_gateway.move

### Function: `add_trusted_emitter(_: &AdminCap, state: &mut GatewayState, emitter_id: u16, emitter: vector<u8>, _ctx: &mut TxContext)`

The `add_trusted_emitter` function adds a new trusted emitter to `GatewayState`.

- [x] The caller must provide a valid `AdminCap`.
- [x] It must ensure that `state.is_initialized` is `true`.
- [x] The `emitter` is added to `state.trusted_emitters` under the key `emitter_id`.

### Function: `remove_trusted_emitter(_: &AdminCap, state: &mut GatewayState, emitter_id: u16, _ctx: &mut TxContext)`

The `remove_trusted_emitter` function removes an emitter from `GatewayState`.

- [x] The caller must provide a valid `AdminCap`.
- [x] It must ensure that `state.is_initialized` is `true`.
- [x] The entry `emitter_id` is removed from `state.trusted_emitters`.

### Function: `add_trusted_receiver(_: &AdminCap, state: &mut GatewayState, receiver_id: u16, receiver: vector<u8>, _ctx: &mut TxContext)`

The `add_trusted_receiver` function adds a new trusted receiver to `GatewayState`.

- [x] The caller must provide a valid `AdminCap`.
- [x] It must ensure that `state.is_initialized` is `true`.
- [x] The `receiver` is added to `state.trusted_receivers` under the key `receiver_id`.

### Function: `remove_trusted_receiver(_: &AdminCap, state: &mut GatewayState, receiver_id: u16, _ctx: &mut TxContext)`

The `remove_trusted_receiver` function removes a receiver from `GatewayState`.

- [x] The caller must provide a valid `AdminCap`.
- [x] It must ensure that `state.is_initialized` is `true`.
- [x] The entry `receiver_id` is removed from `state.trusted_receivers`.

### Function: `pause(_: &AdminCap, state: &mut GatewayState, _ctx: &mut TxContext)`

The `pause` function sets `state.paused` to `true`.

- ☑ The caller must provide a valid `AdminCap`.
- ☑ It must ensure that `state.is_initialized` is `true`.
- ☑ `state.paused` is set to `true`.

#### Function: `unpause(_: &AdminCap, state: &mut GatewayState, _ctx: &mut TxContext)`

The `unpause` function sets `state.paused` to `false`.

- ☑ The caller must provide a valid `AdminCap`.
- ☑ It must ensure that `state.is_initialized` is `true`.
- ☑ `state.paused` is set to `false`.

#### Function: `update_minting_limit(_: &AdminCap, state: &mut GatewayState, new_limit: u64, _ctx: &mut TxContext)`

The `update_minting_limit` function updates the minting limit for `GatewayState`.

- ☑ The caller must provide a valid `AdminCap`.
- ☑ It must ensure that `state.is_initialized` is `true`.
- ☑ `state.minting_limit` is set to `new_limit`.

#### Function: `change_admin(admin_cap: AdminCap, new_admin: address, ctx: &mut TxContext)`

The `change_admin` function transfers `AdminCap` to `new_admin`.

- ☑ The caller must hold the current `AdminCap`.
- ☑ The `admin_cap` is transferred to `new_admin`.

#### Function: `redeem_tokens<CoinType>(state: &mut GatewayState, capabilities: &mut GatewayCapabilities, wormhole_state: &mut WormholeState, treasury: &mut WrappedTokenTreasury<CoinType>, token_bridge_state: &mut token_bridge::state::State, token_state: &mut TBTC::TokenState, vaa_bytes: vector<u8>, clock: &Clock, ctx: &mut TxContext)`

The `redeem_tokens` function verifies the VAA and redeems the wrapped tokens. If the minting limit is not exceeded, tBTC is minted to the `recipient`; otherwise, wrapped coins are transferred directly. The wrapped tokens are stored in `treasury`.

- ☑ The contract must be initialized and not paused.
- ☑ VAA is parsed and verified.

- ☑ It must ensure that the VAA hash is not in `state.processed_vaas`.
- ☑ It must ensure that `emitter_chain` exists in `state.trusted_emitters`.
- ☑ It must ensure that `emitter_address` matches the stored trusted emitter.
- ☑ The VAA hash is recorded in `state.processed_vaas`.
- ☑ Wrapped coins are redeemed via `complete_transfer_with_payload`.
- ☑ If `state.minted_amount + amount <= state.minting_limit`, tBTC is minted to the `recipient`; otherwise, wrapped coins are transferred directly.
- ☑ `state.minted_amount` is updated accordingly.

**Function:** `send_tokens<CoinType>(state: &mut GatewayState, capabilities: &mut GatewayCapabilities, token_bridge_state: &mut token_bridge::state::State, token_state: &mut TBTC::TokenState, treasury: &mut WrappedTokenTreasury<CoinType>, wormhole_state: &mut WormholeState, recipient_chain: u16, recipient_address: vector<u8>, coins: Coin<TBTC::TBTC>, nonce: u32, message_fee: Coin<sui::sui::SUI>, clock: &Clock, ctx: &mut TxContext)`

The `send_tokens` function burns tBTC tokens and publishes a Wormhole message to transfer tBTC tokens to the `recipient` in the `recipient_chain`.

- ☑ The contract must be initialized and not paused.
- ☑ It must ensure that `recipient_chain` exists in `state.trusted_receivers`.
- ☑ The `treasury.tokens` balance must be equal to or larger than the `coins` amount.
- ☑ It burns `coins`.
- ☑ Equivalent wrapped tokens are withdrawn from the treasury.
- ☑ `state.minted_amount` is decreased by the burned amount.
- ☑ Transfer is prepared and the Wormhole message is published.

**Function:** `send_wrapped_tokens<CoinType>(state: &mut GatewayState, capabilities: &mut GatewayCapabilities, token_bridge_state: &mut token_bridge::state::State, wormhole_state: &mut WormholeState, recipient_chain: u16, recipient_address: vector<u8>, coins: Coin<CoinType>, nonce: u32, message_fee: Coin<sui::sui::SUI>, clock: &Clock, _ctx: &mut TxContext)`

The `send_wrapped_tokens` function publishes a Wormhole message to directly transfer wrapped tokens to the `recipient` in the `recipient_chain`.

- ☑ The contract must be initialized and not paused.
- ☑ It must ensure that `recipient_chain` exists in `state.trusted_receivers`.
- ☑ Transfer of wrapped `coins` is prepared and the Wormhole message is published.

# 5. Assessment Results

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

During our assessment on the scoped tBTC Sui Integration contracts, we discovered four findings. No critical issues were found. One finding was of high impact, one was of low impact, and the remaining findings were informational in nature.

## 5.1. 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.