

January 16, 2025

Echelon

Smart Contract Security Assessment

Placeholder text for the Smart Contract Security Assessment report content.

Contents

About Zellic	4
<hr data-bbox="488 403 1565 407"/>	
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
<hr data-bbox="488 785 1565 789"/>	
2. Introduction	6
2.1. About Echelon	7
2.2. Methodology	7
2.3. Scope	9
2.4. Project Overview	10
2.5. Project Timeline	11
<hr data-bbox="488 1226 1565 1230"/>	
3. Detailed Findings	11
3.1. Incorrect integer parsing	12
3.2. Incorrect assertion in deposit_manager	14
3.3. Incorrect calculation of share_proportion	15
3.4. Unchecked deposit amount	17
3.5. Nonce shadowed by echelon_executor	19
<hr data-bbox="488 1671 1565 1675"/>	
4. Discussion	20
4.1. Minor issues with DUST_THRESHOLD	21

4.2.	Share amount could be zero	21
<hr data-bbox="488 403 1567 407"/>		
5.	System Design	22
5.1.	Components: echelon_oracle and initia_deploy	23
5.2.	Components: deposit_manager, echelon_executor, and meridian_executor	23
5.3.	Component: eth_verifier	24
<hr data-bbox="488 724 1567 728"/>		
6.	Assessment Results	24
6.1.	Disclaimer	25

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 Echelon from January 3rd to January 13th, 2025. During this engagement, Zellic reviewed Echelon'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:

- Are there any issues in the signature-verification process?
 - Are there any issues in the deposit and withdrawal process?
 - Are there any issues in the access control of the functions?
-

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. During this assessment, gaps in response times as well as a lack of developer-oriented documentation and thorough end-to-end testing impacted the momentum of our auditors.

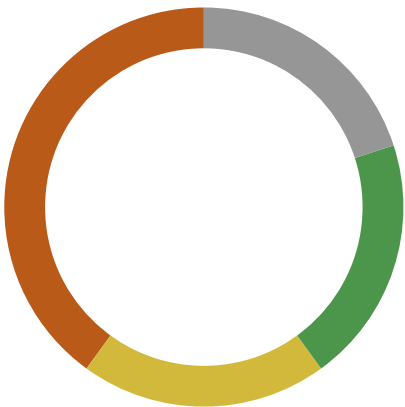
1.4. Results

During our assessment on the scoped Echelon contracts, we discovered five findings. No critical issues were found. Two findings were of high impact, one was of medium impact, one was of low impact, and the remaining finding was informational in nature.

Additionally, Zellic recorded its notes and observations from the assessment for the benefit of Echelon in the Discussion section ([4.7](#)).

Breakdown of Finding Impacts

Impact Level	Count
<div>Critical</div>	0
<div>High</div>	2
<div>Medium</div>	1
<div>Low</div>	1
<div>Informational</div>	1



2. Introduction

2.1. About Echelon

Echelon contributed the following description of Echelon:

Echelon is an advanced money market written in Move designed to unlock yield access at scale. Echelon is the flagship lending product for crypto, BTCfi, RWA, and CeDeFi across Initia, Movement and Aptos, and the Ethena Network. Echelon adapts best practices from AAVEv3, Fluid, Fraxlend, and Radiant to provide a capital efficient and feature rich lending experience for yield bearing assets.

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. 7) 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:

Echelon Contracts

Type	Move
Platform	Aptos
Target	echelon-modules
Repository	https://github.com/EchelonMarket/echelon-modules ↗
Version	e207b94e972787b3a145c7d4a4e7b20230a1ba46
Programs	echelon_oracle/sources/oracle.move initia_deploy/sources/factory.move
Target	royco-move
Repository	https://github.com/EchelonMarket/royco-move ↗
Version	e6b4ecbcde6fd0ec50e0580a5f01b53f4529ccc3
Programs	echelon_executor/sources/executor.move deposit_manager/sources/controller.move meridian_executor/sources/executor.move

Target	evm-signature-verifier
Repository	https://github.com/EchelonMarket/evm-signature-verifier ↗
Version	4d90631229f590d3e9e004111a6dea5499838f77
Programs	eth_verifier.move

2.4. Project Overview

Zellic was contracted to perform a security assessment for a total of 2.1 person-weeks. The assessment was conducted by two consultants over the course of two calendar weeks.


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:


Sunwoo Hwang
 Engineer
sunwoo@zellic.io ↗


Juchang Lee
 Engineer
lee@zellic.io ↗

2.5. Project Timeline

The key dates of the engagement are detailed below.

January 3, 2025	Kick-off call
<hr data-bbox="490 508 1062 512"/>	
January 3, 2025	Start of primary review period
<hr data-bbox="490 588 1062 592"/>	
January 13, 2025	End of primary review period

3.1. Incorrect integer parsing

Target	controller.move		
Category	Coding Mistakes	Severity	High
Likelihood	High	Impact	High

Description

The `parse_deposit_payload` function incorrectly handles integer-value parsing. While Solidity stores integer values into memory in big-endian format (right to left), the `from_bcs` module parses integers in little-endian format (left to right).

```
// Per Payload (first 65 bytes):
let market_hash = from_bcs::(vector::slice(&message, 0, 32));
let ccdm_nonce = from_bcs::(vector::slice(&message, 32, 64));
let expected_messages = from_bcs::(vector::slice(&message, 64, 65));

// Per Depositor (following 32 byte blocks):
let depositor_indices = vector::range_with_step(65, vector::length(&message),
    32);
let depositor_map = simple_map::new();
vector::for_each(depositor_indices, |index| {
    assert!(index + 32 <= vector::length(&message),
        ERR_DEPOSIT_MANAGER_MALFORMED_PAYLOAD);
    let depositor_address = vector::slice(&message, index, index + 20);
    let depositor_amount = from_bcs::(vector::slice(&message, index +
        20, index + 32));
    assert!(depositor_amount <= MAX_U64,
        ERR_DEPOSIT_MANAGER_DEPOSIT_AMOUNT_OVERFLOW);

    let depositor_amount_adjusted = ((depositor_amount / (math64::pow(10,
        asset_mantissa_diff) as u128)) as u64);
    let amount_vector = vector[depositor_amount_adjusted];
    simple_map::add(&mut depositor_map, depositor_address, amount_vector);
});
```

This is the difference of the results between the controller test code and the Solidity code. The address is the same, but the `ccdm_nonce` in Solidity would be `0x100...0` in the module.

[illegible]

```

hash
0100000000000000000000000000000000000000000000000000000000000000 // ccdm nonce

// CCDMPayloadLib.sol
0000000000000000000000000000000000000000000000000000000000000000123456789 // market
hash
0000000000000000000000000000000000000000000000000000000000000001 // ccdm nonce

```

Impact

The incorrect integer parsing can result in misinterpreted deposit amounts and invalid nonce values.

Recommendations

Reverse the byte order of integer values before passing values to the `from_bcs` module to ensure proper integer parsing.

Remediation

This issue has been acknowledged by Echelon, and a fix was implemented in commit [f6fc03c0](#).

3.2. Incorrect assertion in deposit_manager

Target	controller.move		
Category	Coding Mistakes	Severity	High
Likelihood	High	Impact	High

Description

In the process of validating the message, there is an assertion that always fails.

While the message length is always greater than 65 bytes, taking the modulo of the length by 32 will always result in a value less than 32, making it impossible to equal 65.

```

fun parse_deposit_payload(received_asset_metadata: Object<Metadata>,
    received_asset_amount: u64, message: vector<u8>):
    DepositPayload acquires DepositManagerController {
    // [...]
    assert!(vector::length(&message) % 32 == 65,
        ERR_DEPOSIT_MANAGER_MALFORMED_PAYLOAD);
    // [...]
    }

```

Impact

Regardless of the validity of the message, parse_deposit_payload will always cause a revert.

Recommendations

Make a different assertion to properly validate the message.

Remediation

This issue has been acknowledged by Echelon, and a fix was implemented in commit [c368b175](#).

3.3. Incorrect calculation of share_proportion

Target	executor.move		
Category	Coding Mistakes	Severity	Medium
Likelihood	High	Impact	Medium

Description

An incorrect calculation exists when calculating share_proportion in execute_deposit.

As of the time of writing, the calculation is $\text{asset_amounts} * \text{BPS_BASE} / \text{deposit_amounts}$. However, this calculation will not properly calculate the percentage of the deposit. In the current situation, the number of users is divided by the total number of tokens, so the exact ratio is not calculated.

```
// meridian_executor/executor.move
public entry fun execute_deposit(
    // Standard deposit params
    nonce: u256,
    market_hash: address,
    // Meridian specific params
    pool_obj: Object<Pool>
) acquires ExecutorController, SmartSigner {
    // [...]
    vector::for_each(simple_map::keys(&depositor_map),
        |depositor_eth_address| {
            // Determine the proportion of the deposit that the user contributed to
            // the pool
            let deposit_amounts = *simple_map::borrow(&depositor_map,
                &depositor_eth_address);
            let share_proportion = math64::mul_div(*vector::borrow(&asset_amounts,
                0), BPS_BASE, *vector::borrow(&deposit_amounts, 0)); // NOTE: the
            // proportion should be the same for first vs. second asset
            let depositor_lp_token_share = math64::mul_div(lp_token_amount,
                share_proportion, BPS_BASE);
            let depositor_refund_shares = vector::map_ref(&refund_assets_amounts,
                |amount| math64::mul_div(*amount, share_proportion, BPS_BASE));
            // [...]
        });
    // [...]
}
```

Impact

Due to the incorrect calculation, `depositor_lp_token_shares` and `depositor_refund_shares` are also incorrectly calculated. This means that the correct values will not be deposited.

Recommendations

For an accurate calculation, it should be `deposit_amounts * BPS_BASE / asset_amounts`.

Remediation

This issue has been acknowledged by Echelon, and a fix was implemented in commit [03b432fd](#).

3.4. Unchecked deposit amount

Target	controller.move		
Category	Coding Mistakes	Severity	Low
Likelihood	Low	Impact	Low

Description

The `receive_deposit_batch` function calls `parse_deposit_payload` to parse deposit payloads. While `received_asset_amount` represents the total sum of all depositor amounts, the `parse_deposit_payload` function lacks validation to ensure this sum matches the actual total deposit amount received.

```

fun parse_deposit_payload(received_asset_metadata: Object<Metadata>,
    received_asset_amount: u64, message: vector<u8>):
    DepositPayload acquires DepositManagerController {
        // [...]
        let asset_mantissa_diff =
            *smart_table::borrow(&controller.asset_decimals_mantissa_table,
                received_asset_metadata);

        // [...]
        let depositor_indices = vector::range_with_step(65,
            vector::length(&message), 32);
        let depositor_map = simple_map::new();
        vector::for_each(depositor_indices, |index| {
            assert!(index + 32 <= vector::length(&message),
                ERR_DEPOSIT_MANAGER_MALFORMED_PAYLOAD);
            let depositor_address = vector::slice(&message, index, index + 20);
            let depositor_amount = from_bcs::to_u128(vector::slice(&message,
                index + 20, index + 32));
            assert!(depositor_amount <= MAX_U64,
                ERR_DEPOSIT_MANAGER_DEPOSIT_AMOUNT_OVERFLOW);

            let depositor_amount_adjusted = ((depositor_amount / (math64::pow(10,
                asset_mantissa_diff) as u128)) as u64);
            let amount_vector = vector[depositor_amount_adjusted];
            simple_map::add(&mut depositor_map, depositor_address,
                amount_vector);
        });
        // [...]
    }

```

Impact

The lack of validation between the received total amount and the sum of individual deposits could potentially allow inconsistent deposit states. While this does not directly lead to fund loss, it may cause accounting discrepancies that need to be reconciled manually.

Recommendations

Since there is logic that adjusts deposit amounts based on asset decimals, the `received_asset_amount` may not exactly match the total sum of deposits. We recommend adding a validation check to ensure the sum of all depositor amounts approximately matches the `received_asset_amount`.

Remediation

This issue has been acknowledged by Echelon, and a fix was implemented in commit [d1afe83e7](#).

3.5. Nonce shadowed by echelon_executor

Target	echelon_executor.move		
Category	Coding Mistakes	Severity	Informational
Likelihood	N/A	Impact	Informational

Description

The nonce of the execute_deposit function is shadowed. However, this will eventually be set to the appropriate nonce value, so there are no program flow issues.

```
// echelon_executor/executor.move
public entry fun execute_deposit(
    nonce: u256,
    market_hash: address,
    market_obj: Object<Market>
) acquires ExecutorController, SmartSigner {
    // [...]
    let nonce =
        borrow_global<ExecutorController>(package::package_address()).last_nonce +
        1;
    // [...]
}
```

In addition, the nonce checks are different between meridian_executor and echelon_executor. However, this gives the same result, so there are no program flow issues.

```
// meridian_executor/executor.move
public entry fun execute_deposit(
    nonce: u256,
    market_hash: address,
    pool_obj: Object<Pool>
) acquires ExecutorController, SmartSigner {
    // [...]
    assert!(
        borrow_global<ExecutorController>(package::package_address()).last_nonce +
        1 == nonce, ERR_MERIDIAN_EXECUTOR_INVALID_NONCE
    );
    // [...]
}
```

Impact

Currently, this structure makes it difficult for future nonce-related changes to be processed in batches, which can be unmanageable and cause changes to be missed. This means that if the nonce-related structure changes in the future, the contract may not work properly due to inconsistent nonce handling.

Recommendations

It is recommended to avoid shadowing and make the nonce checks consistent between the two contracts.

Remediation

This issue has been acknowledged by Echelon, and a fix was implemented in commit [3389f079](#).

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. Minor issues with DUST_THRESHOLD

In `meridian_executor/executor.move`, the `execute_deposit` function will send the amount to the admin if the remaining balance after processing is less than `DUST_THRESHOLD`.

This is not a security issue, but a user can consider this is a loss of funds. so Echelon should be given ample notice.

Also, since `DUST_THRESHOLD` is set to 1000 while `BPS_AMOUNT` is 10000, the precision error can be up to 10,000 or less. We recommend setting the `DUST_THRESHOLD` equal to `BPS_AMOUNT` to prevent this issue.

```
vector::for_each(refund_assets, |asset| {
    assert!(fungible_asset::amount(&asset) < DUST_THRESHOLD,
        ERR_MERIDIAN_EXECUTOR_TOO_MUCH_ASSET_SUPPLIED);
    primary_fungible_store::deposit(manager::manager(), asset);
});
```

4.2. Share amount could be zero

The `execute_deposit` function in `meridian_executor::executor` adds deposited tokens to the pool and shares the LP token with the depositor based on the proportion of the total deposit amount. The proportion of the LP token share is calculated as follows:

```
share_proportion = deposit_amount * BPS_BASE / total_deposit_amount
```

When the `total_deposit_amount` is greater than `deposit_amount * BPS_BASE`, the `share_proportion` will be zero, resulting in the depositor receiving no LP token shares. For example, if one user deposits 100 USDC while 100 other users each deposit 10,000 USDC, the share proportion will be zero.

```
// Process each depositor's deposit, creating a smart account with proportional
// LP tokens and refund assets for each
vector::for_each(simple_map::keys(&depositor_map), |depositor_eth_address| {
    // Determine the proportion of the deposit that the user contributed to the
    // pool
```

```
let deposit_amounts = *simple_map::borrow(&depositor_map,
&depositor_eth_address);
let share_proportion = math64::mul_div(*vector::borrow(&asset_amounts, 0),
BPS_BASE, *vector::borrow(&deposit_amounts, 0)); // NOTE: the proportion
should be the same for first vs. second asset
let depositor_lp_token_share = math64::mul_div(lp_token_amount,
share_proportion, BPS_BASE);
let depositor_refund_shares = vector::map_ref(&refund_assets_amounts,
|amount| math64::mul_div(*amount, share_proportion, BPS_BASE));

// Extract LP tokens & refunds from batched assets into the depositor's
smart signer
let smart_signer = generate_smart_signer(depositor_eth_address);
primary_fungible_store::deposit(signer::address_of(&smart_signer),
fungible_asset::extract(&mut lp_token, depositor_lp_token_share));
vector::enumerate_mut(&mut refund_assets, |index, asset| {
    let refund_amount = *vector::borrow(&depositor_refund_shares, index);
    primary_fungible_store::deposit(signer::address_of(&smart_signer),
fungible_asset::extract(asset, refund_amount));
});
});
```

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. Components: echelon_oracle and initia_deploy

Description

The `initia_deploy::factory` module is responsible for deploying and managing extend references for new Move packages. It publishes and stores code for newly deployed modules on chain and generates signer permissions for newly created on-chain objects, enabling them to act as signers in subsequent transactions.

The `echelon_oracle::oracle` module is responsible for storing and retrieving on-chain price information for assets. It maps each fungible asset's metadata to an oracle price identifier and retrieves and validates price data from a general on-chain oracle, ensuring freshness of the price via a max-staleness constraint.

Invariants

For `initia_deploy::factory`, only a valid object signer can retrieve the object's `ExtendRef`.

For `echelon_oracle::oracle`, price-feed updates are only allowed by an entity recognized as a manager. The max-staleness check ensures that no stale price data can be used.

5.2. Components: deposit_manager, echelon_executor, and meridian_executor

Description

The `deposit_manager::controller` module is responsible for receiving cross-chain deposits and storing them. It parses deposit messages to extract depositor addresses and amounts. The valid executors can pull deposits from the `deposit_manager` and supply the assets to the market.

The `echelon_executor::executor` module is responsible for pulling deposits from `deposit_manager` and supplying the assets to the market. The users who have valid signatures of Ethereum addresses can withdraw the assets from the market.

The `meridian_executor::executor` module is responsible for pulling deposits from `deposit_manager`, supplying the assets to the pool, and distributing LP tokens to depositors. The users who have valid signatures of Ethereum addresses can withdraw the assets from the pool.

Invariants

For `deposit_manager::controller`, only a valid executor can pull deposits from the `deposit_manager`.

For `echelon_executor::executor`, only users who have valid signatures of Ethereum addresses can withdraw the assets from the market.

For `meridian_executor::executor`, only users who have valid signatures of Ethereum addresses can withdraw the assets from the pool.

5.3. Component: `eth_verifier`

Description

The `eth_verifier` component is responsible for linking an Ethereum address to a specific Aptos account by verifying an Ethereum signature against an Aptos address. It verifies that a provided Ethereum address corresponds to the holder of a specific Aptos address. In other words, it ensures that the Ethereum address in question has signed a message containing the Aptos address, thus authenticating that the signer on Ethereum is indeed the same entity controlling the Aptos account.

Invariants

For `eth_verifier`, there are two invariants.

1. **Valid signature length.** The signature must be exactly 64 bytes. This is enforced by an assert that checks the byte length of the signature.
2. **Correct signing.** The recovered Ethereum address from the signature must match the `eth_address` argument provided in the function call. If not, the code aborts with `ERR_ETH_ADDRESS_MISMATCH`. The Aptos address derived from the BCS-encoded message must match the `signer::address_of(user)`. If not, the code aborts with `ERR_APTOS_ADDRESS_MISMATCH`.

6. Assessment Results

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

During our assessment on the scoped Echelon contracts, we discovered five findings. No critical issues were found. Two findings were of high impact, one was of medium impact, one was of low impact, and the remaining finding was informational in nature.

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.