Zellic

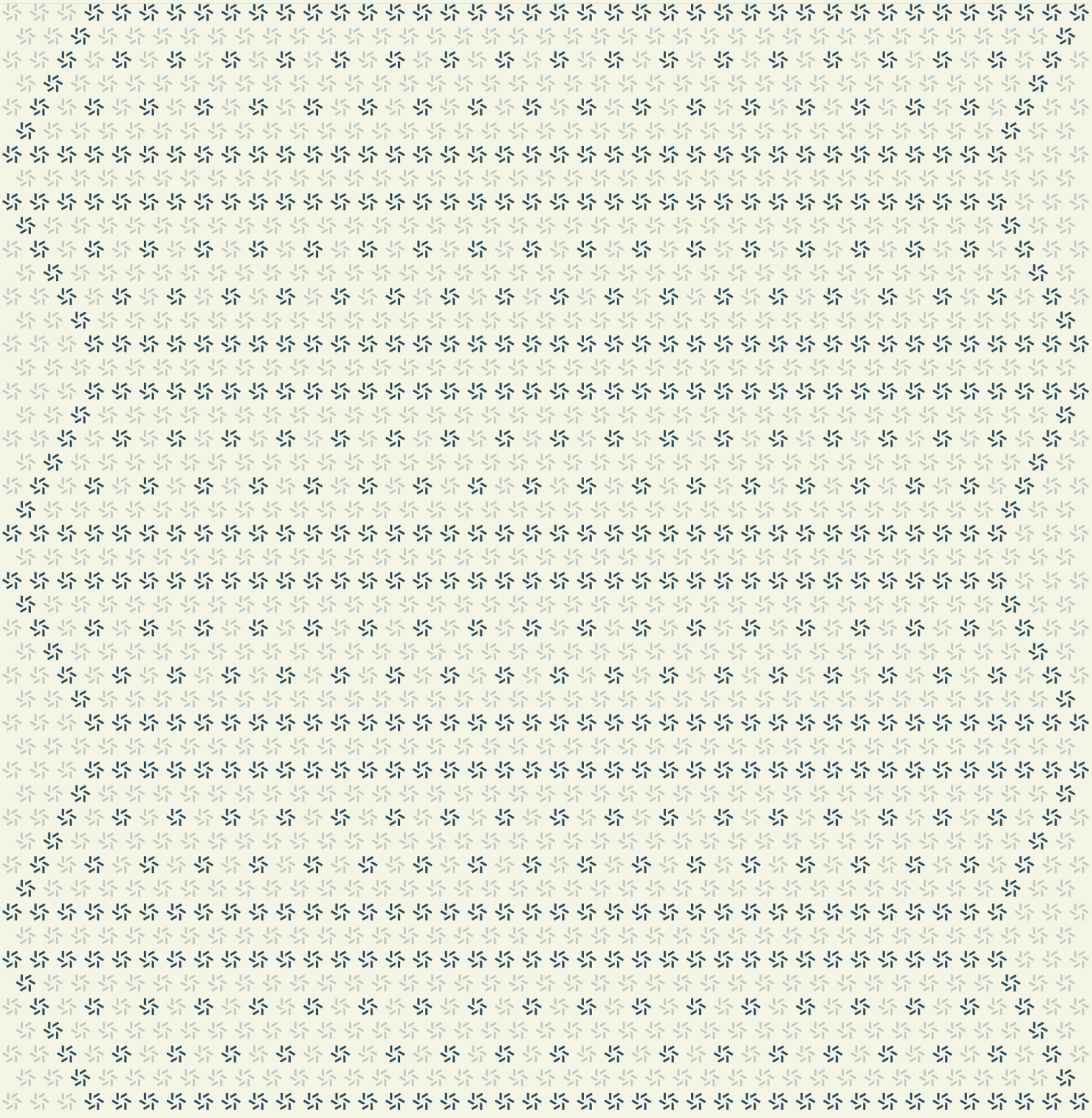
Prepared for
Fernando Otero
Anza Technology, Inc.

Prepared by
Nathanial Lattimer
Maik Robert
Avraham Weinstock
Zellic

June 30, 2025

Pinocchio and p-token

Solana 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 Anza Technology, Inc. from May 27th to June 16th, 2025. During this engagement, Zellic reviewed Pinocchio and p-token'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:

- For Pinocchio and p-token, are there any undefined behaviors or unsafe memory access?
- For p-token, does the program exhibit the exact same behavior as the existing SPL Token program?

1.3. Non-goals and Limitations

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

- Components outside of the mentioned folders
- Infrastructure relating to the project

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 Pinocchio library and p-token program, we discovered eight findings. One critical issue was found. Three were of high impact, one was of medium impact, and three were of low impact.

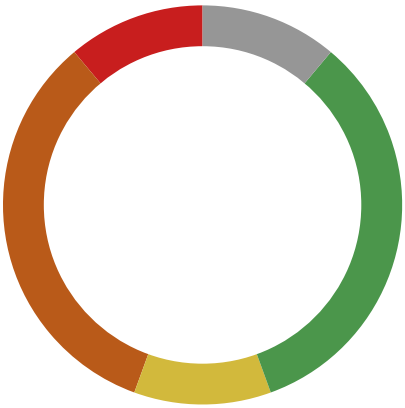
Additionally, Zellic recorded its notes and observations from the assessment for the benefit of Anza Technology, Inc. in the Discussion section (4. ↗).

Based on the number of severe findings uncovered during the audit, it is our opinion that the project is not yet ready for production. We strongly advise a comprehensive reassessment before deployment to help identify any potential issues or vulnerabilities introduced by necessary fixes or changes. We also recommend adopting a security-focused development workflow, including (but not limited to) augmenting the repository with comprehensive end-to-end tests that achieve 100%

branch coverage using any common, maintainable testing framework, thoroughly documenting all function requirements, and training developers to have a security mindset while writing code.

Breakdown of Finding Impacts

Impact Level	Count
Critical	1
High	3
Medium	1
Low	3
Informational	1



2. Introduction

2.1. About Pinocchio and p-token

Anza Technology, Inc. contributed the following description of Pinocchio and p-token:

> Pinocchio is a zero-dependency library to create Solana programs in Rust. It takes advantage of the way SVM loaders serialize the program input parameters into a byte array that is then passed to the program's entrypoint to define zero-copy types to read the input. Since the communication between a program and SVM loader is done via a byte array, Pinocchio defines its own types to mitigate dependency issues.
>
> p-token is a reimplementation of the SPL Token program using Pinocchio. The purpose is to have an implementation that optimizes the compute units, while being fully compatible with the original implementation — i.e., support the exact same instruction and account layouts as SPL Token, byte for byte.

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 programs.

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 programs 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:

Pinocchio and p-token Programs

Type	Rust
Platform	Solana
Target	pinocchio
Repository	https://github.com/anza-xyz/pinocchio ↗
Version	`887a1150bba40c5c77373da90b76557d8c73b747`

Programs	`sdk/pinocchio/*`
	`sdk/log/*`
	`sdk/pubkey/*`
	`programs/associated-token-account/*`
	`programs/memo/*`
	`programs/system/*`
	`programs/token/*`

Target	token
Repository	https://github.com/solana-program/token ↗
Version	`4c3fabd685e30b27bbc8f5d515626177dd720a02`
Programs	`p-token/src/*` `interface/*`

Target	PR review for PR#176
Repository	https://github.com/anza-xyz/pinocchio ↗
Version	`6e77c1ca228ec9cd79cd2ea97501ecef8e899fc0`
Programs	`sdk/pinocchio/src/entrypoint/mod.rs`

Target	PR review for PR#209
Repository	https://github.com/anza-xyz/pinocchio ↗
Version	`8cbac4eec3093692ca1fc3a0d07c8d8696bf3031`
Programs	`sdk/pinocchio/src/entrypoint/mod.rs`

Target	PR review for PR#80
Repository	https://github.com/solana-program/token ↗
Version	`ae48e9943e04a7adbcbd142945ecdc111890620d`
Programs	`src/processor/batch.rs` `tests/batch.rs`

Target	PR review for PR#81
Repository	https://github.com/solana-program/token ↗
Version	`0fe8e2cdb120e03bd408da1b6d9167ba193586c5`

Programs	`p-interface/src/lib.rs`
	`p-interface/src/state/account.rs`
	`p-token/src/processor/mod.rs`
	`p-token/src/processor/shared/transfer.rs`

Target	PR review for PR#82
Repository	https://github.com/solana-program/token ↗
Version	`7941c3502d0a147194b8b29d36ea2b47a28204bc`
Programs	`p-token/src/processor/shared/transfer.rs`

2.4. Project Overview

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

Nathanial Lattimer
Engineer
d0nut@zellic.io

Maik Robert
Engineer
maik@zellic.io

Avraham Weinstock
Engineer
avi@zellic.io

2.5. Project Timeline

The key dates of the engagement are detailed below.

May 27, 2025	Kick-off call
May 27, 2025	Start of primary review period
June 16, 2025	End of primary review period
September 10, 2025	End of PR review period

3. Detailed Findings

3.1. Dangerous use of `?Sized` bound on `copy_val` leads to out-of-bounds read

Target	sdk/pinocchio/src/memory.rs		
Category	Coding Mistakes	**Severity**	Critical
Likelihood	High	**Impact**	Critical

Description

The `copy_val` function takes two parameters of generic type `T`. And `T` has the bound `?Sized`, which indicates that `T` may not necessarily implement the `Sized` trait. This means that `T` may not have static size.

```
#[inline]
pub fn copy_val<T: ?Sized>(dst: &mut T, src: &T) {
    #[cfg(target_os = "solana")]
    // SAFETY: dst and src are of same type therefore the size is the same
    unsafe {
        syscalls::sol_memcpy_(
            dst as *mut T as *mut u8,
            src as *const T as *const u8,
            core::mem::size_of_val(dst) as u64,
        );
    }

    #[cfg(not(target_os = "solana"))]
    core::hint::black_box((dst, src));
}
```

As a result, two instances of `T` may have differing sizes. The `copy_val` function uses the size of the first parameter, `dst`, to determine how much of `src` to copy over. Because these instances of `T` may differ in size, it can result in the situation where bytes are read past the end of `src` and into potentially uninitialized memory or other fields.

Impact

This may lead to an out-of-bounds read against `src`. Reading out-of-bounds may leak sensitive data not intended to be read by the program, or otherwise lead to undefined behavior by reading bytes that were not meant to be read.

Recommendations

Remove the `?Sized` constraint as `T` will assume `Sized` by default.

Remediation

This issue has been acknowledged by Anza Technology, Inc., and a fix was implemented in commit 293cc13c ↗.

3.2. Missing `Copy` bound on `copy_val` function

Target	sdk/pinocchio/src/memory.rs		
Category	Coding Mistakes	**Severity**	High
Likelihood	High	**Impact**	High

Description

The `copy_val` function takes two parameters of generic type `T` and copies bytes from `src` to `dst`.

```
#[inline]
pub fn copy_val<T: ?Sized>(dst: &mut T, src: &T) {
    #[cfg(target_os = "solana")]
    // SAFETY: dst and src are of same type therefore the size is the same
    unsafe {
        syscalls::sol_memcpy_(
            dst as *mut T as *mut u8,
            src as *const T as *const u8,
            core::mem::size_of_val(dst) as u64,
        );
    }

    #[cfg(not(target_os = "solana"))]
    core::hint::black_box((dst, src));
}
```

At first glance, this operation seems harmless, but it has the potential to break important invariants. For example, imagine a type that internally stores a unique identifier and intentionally does not implement `Clone` nor `Copy`. This function would allow creating two instances of that type with the same unique identifier, breaking the invariant.

In Rust, the `Copy` trait is used to indicate that a byte-by-byte copy of a type is a legal and safe operation. Therefore, for `copy_val` to be safe, it should require the `Copy` trait as a bound on `T`. This also has the additional effect of requiring that the type does not have a custom `Drop` implementation, meaning that it is not required to drop the instance in place before writing over `dst`.

Impact

This can be used to break invariants that are typically enforced on types without the `Copy` trait.

Recommendations

Require that `T` implements `Copy`.

Remediation

This issue has been acknowledged by Anza Technology, Inc., and a fix was implemented in commit 293cc13c ↗.

3.3. The `Transmutable` trait is missing `unsafe` due to memory safety requirements

Target	interface/src/state/mod.rs		
Category	Coding Mistakes	**Severity**	High
Likelihood	Medium	**Impact**	High

Description

The `Transmutable` trait in `interface` has memory safety implications in its implementation. Types that implement the trait are instructed to ensure that casts from raw bytes are safe.

```
/// Marker trait for types that can be cast from a raw pointer.
///
/// It is up to the type implementing this trait to guarantee that the cast is
/// safe, i.e., the fields of the type are well aligned and there are no padding
/// bytes.
pub trait Transmutable {
    /// The length of the type.
    ///
    /// This must be equal to the size of each individual field in the type.
    const LEN: usize;
}
```

This requires understanding the alignment of the underlying type, ensuring that the type does not introduce padding, that arbitrary byte representations are legal, and so on. Given the memory safety requirements of this trait, the trait should be marked `unsafe`.

Impact

Undefined behavior can result from an incorrect implementation of `Transmutable`, despite the trait not being an `unsafe` trait.

Recommendations

Mark `Transmutable` as `unsafe`.

Remediation

This issue has been acknowledged by Anza Technology, Inc., and a fix was implemented in commit 5ac906f0 ↗.

3.4. Log trait missing `unsafe`

Target	sdk/log/crate/src/logger.rs		
Category	Coding Mistakes	**Severity**	High
Likelihood	Medium	**Impact**	High

Description

The `Log` trait in the log crate is used to mark a type as loggable. It requires the implementer to provide the implementation of (at least) `write_with_args`, which takes a buffer of `MaybeUninit<u8>` and parameters relevant for logging. It then returns the amount of bytes that it had written to.

```
/// Trait to specify the log behavior for a type.
pub trait Log {
    #[inline(always)]
    fn debug(&self, buffer: &mut [MaybeUninit<u8>]) -> usize {
        self.debug_with_args(buffer, &[])
    }

    #[inline(always)]
    fn debug_with_args(&self, buffer: &mut [MaybeUninit<u8>], args:
    &[Argument]) -> usize {
        self.write_with_args(buffer, args)
    }

    #[inline(always)]
    fn write(&self, buffer: &mut [MaybeUninit<u8>]) -> usize {
        self.write_with_args(buffer, &[])
    }

    fn write_with_args(&self, buffer: &mut [MaybeUninit<u8>], parameters: &[
        Argument]) -> usize;
}
```

This buffer, passed from the `Logger` struct, is ultimately then read from to log the message.

The implementation of `write_with_args` has memory safety implications. If a `usize` is returned that does *not* accurately reflect the number of bytes consecutively written to the buffer, this can result in undefined behavior by reading from an uninitialized `MaybeUninit<u8>`. In Rust, if the implementation of a trait can result in undefined behavior, it must be marked with `unsafe` to

communicate in the type signature that the implication of this trait has memory safety implications.

Impact

Undefined behavior can result from an incorrect implementation of `Log`, despite the trait not being an `unsafe` trait. This could result in reading uninitialized memory in otherwise safe Rust.

Recommendations

Either mark `Log` as `unsafe`, or change the signature to use `&[u8]`.

Remediation

This issue has been acknowledged by Anza Technology, Inc., and a fix was implemented in commit dee49f59 ↗.

3.5. Undefined behavior due to invalid reference casting in `AccountInfo::assign`

Target	sdk/pinocchio/src/account_info.rs		
Category	Coding Mistakes	**Severity**	Medium
Likelihood	Medium	**Impact**	Medium

Description

The `assign` function casts a reference to a `const ptr` and then to a `mut ptr`, before writing to it.

```
#[inline(always)]
pub unsafe fn assign(&self, new_owner: &Pubkey) {
    #[allow(invalid_reference_casting)]
    core::ptr::write_volatile(&(*self.raw).owner as *const _ as *mut Pubkey,
    *new_owner);
}
```

Casting an immutable reference `&` to a mutable one `&mut`, or a `mut ptr`, is always undefined behavior, as pointed out by the Rustonomicon ↗. Here the `#[allow(invalid_reference_casting)]` attribute is used to silence the error the compiler throws otherwise, indicating the undefined behavior.

Impact

This may lead to undefined behavior resulting from casting an immutable reference to a mutable pointer.

Recommendations

Avoid taking an immutable reference, and directly cast a mutable reference to the mutable pointer first. Additionally, consider using `core::ptr::write(..)` over `write_volatile` as there is no direct need for a volatile write.

Remediation

This issue has been acknowledged by Anza Technology, Inc., and a fix was implemented in commit d043b5d6 ↗.

3.6. Incorrectly requiring accounts to be a signer

Target	System program		
Category	Coding Mistakes	**Severity**	Low
Likelihood	High	**Impact**	Low

Description

The new implementation of the system program incorrectly requires one of the accounts passed to the `AllocateWithSeed` and `AssignWithSeed` instructions to be signers.

In the new implementation, both accounts passed to the instruction are required to be signers of the transaction.

Here is an example for `AllocateWithSeed`:

```
pub fn invoke_signed(&self, signers: &[Signer]) -> ProgramResult {
        // account metadata
        let account_metas: [AccountMeta; 2] = [
            AccountMeta::writable_signer(self.account.key()),
            AccountMeta::readonly_signer(self.base.key()),
        ];
        ...
```

Here is an example for `AssignWithSeed`:

```
pub fn invoke_signed(&self, signers: &[Signer]) -> ProgramResult {
        // account metadata
        let account_metas: [AccountMeta; 2] = [
            AccountMeta::writable_signer(self.account.key()),
            AccountMeta::readonly_signer(self.base.key()),
        ];
        ...
```

The current implementation of the system program does not require both accounts to be signers of the transaction.

Here is the `allocate_with_seed` example:

```
pub fn allocate_with_seed(
    address: &Pubkey, // must match create_with_seed(base, seed, owner)
```

```
    base: &Pubkey,
    seed: &str,
    space: u64,
    owner: &Pubkey,
) -> Instruction {
    let account_metas = vec![
        AccountMeta::new(*address, false),
        AccountMeta::new_readonly(*base, true),
    ];
    ...
```

Here is the `assign_with_seed` example:

```
#[cfg(feature = "bincode")]
pub fn assign_with_seed(
    address: &Pubkey, // must match create_with_seed(base, seed, owner)
    base: &Pubkey,
    seed: &str,
    owner: &Pubkey,
) -> Instruction {
    let account_metas = vec![
        AccountMeta::new(*address, false),
        AccountMeta::new_readonly(*base, true),
    ];
    ...
```

Impact

The new implementation would break programs that rely on the correct signer flags to be set.

Recommendations

Adjust the signer flags so they match the current implementation of the system program.

Remediation

This issue has been acknowledged by Anza Technology, Inc., and a fix was implemented in commit fdf3d5fc ↗.

3.7. Undefined behavior due to `MaybeUninit` initialization invariant

Target	Token program		
Category	Coding Mistakes	**Severity**	Low
Likelihood	Medium	**Impact**	Low

Description

The token program makes use of `MaybeUninit` in almost all of its functions. In the type's documentation, we find ways that incorrect usage can lead to undefined behavior in the form of an initialization invariant ↗. This happens when uninitialized memory is read from without first being written to.

In the `invoke_signed` functions of `initialize_mint`, `initialize_mint2`, and `set_authority`, this is the case.

```
pub fn invoke_signed(&self, signers: &[Signer]) -> ProgramResult {
    // Account metadata
    let account_metas: [AccountMeta; 2] = [
        AccountMeta::writable(self.mint.key()),
        AccountMeta::readonly(self.rent_sysvar.key()),
    ];

    // Instruction data layout:
    // -  [0]: instruction discriminator (1 byte, u8)
    // -  [1]: decimals (1 byte, u8)
    // -  [2..34]: mint_authority (32 bytes, Pubkey)
    // -  [34]: freeze_authority presence flag (1 byte, u8)
    // -  [35..67]: freeze_authority (optional, 32 bytes, Pubkey)
    let mut instruction_data = [UNINIT_BYTE; 67]; <--
    MaybeUninit uninitialized bytes

    // Set discriminator as u8 at offset [0]
    write_bytes(&mut instruction_data, &[0]);
    // Set decimals as u8 at offset [1]
    write_bytes(&mut instruction_data[1..2], &[self.decimals]);
    // Set mint_authority as Pubkey at offset [2..34]
    write_bytes(&mut instruction_data[2..34], self.mint_authority);
    // Set COption & freeze_authority at offset [34..67]
    if let Some(freeze_auth) = self.freeze_authority {
        write_bytes(&mut instruction_data[34..35], &[1]);
        write_bytes(&mut instruction_data[35..], freeze_auth);
```

```
    } else {
        write_bytes(&mut instruction_data[34..35], &[0]); <--
    bytes 35-67 have not been written to
    }

    let instruction = Instruction {
        program_id: &crate::ID,
        accounts: &account_metas,
        data: unsafe { from_raw_parts(instruction_data.as_ptr() as _, 67) },
    <-- unitialized bytes 35-67 are being read from
    };

    invoke_signed(&instruction, &[self.mint, self.rent_sysvar], signers)
}
```

In the annotated example above, which is taken from `InitializeMint`, we see that the `instruction_data` is a slice of `UNINIT_BYTE`, which is defined as

```
const UNINIT_BYTE: MaybeUninit<u8> = MaybeUninit::<u8>::uninit();
```

Not all of the uninitialized bytes are written to in both branches of the if statement. In the `else` branch, bytes 35–67 are not written to and are subsequently read from in the `unsafe {}` block, leading to undefined behavior.

Impact

This may lead to undefined behavior due to reading from uninitialized memory that was instantiated using the `MaybeUninit` crate.

Recommendations

Instead of using `MaybeUninit`, it should be preferred to use a slice of `[0u8; X]` instead, which will prevent the undefined behavior. Alternatively, changing the size of the slice in response to the amount written will also avoid the undefined behavior.

Remediation

This issue has been acknowledged by Anza Technology, Inc., and a fix was implemented in commit 32c13da7 ↗.

3.8. Missing alignment safety message

Target	sdk/pinocchio/src/entrypoint/lazy.rs		
Category	Coding Mistakes	**Severity**	Low
Likelihood	Low	**Impact**	Low

Description

The safety comment for `InstructionContext::new_unchecked` reads as follows:

```
/// Creates a new [`InstructionContext`] for the input buffer.
///
/// # Safety
///
/// The caller must ensure that the input buffer is valid, i.e., it represents
/// the program input parameters serialized by the SVM loader.
#[inline(always)]
pub unsafe fn new_unchecked(input: *mut u8) -> Self {
    Self {
        // SAFETY: The first 8 bytes of the input buffer represent the
        // number of accounts when serialized by the SVM loader, which is read
        // when the context is created.
        buffer: unsafe { input.add(core::mem::size_of::<u64>()) },
        // SAFETY: Read the number of accounts from the input buffer serialized
        // by the SVM loader.
        remaining: unsafe { *(input as *const u64) },
    }
}
```

It states that the caller must ensure that the input buffer is valid. To increase clarity, we suggest mentioning that the buffer needs to be eight-byte aligned as this is an additional requirement necessary for casting `*mut u8` to `*const u64`.

Impact

While the safety comment asks the caller to ensure the input buffer is valid, it lacks important context on what constitutes validity.

Recommendations

We recommend including clarification around the alignment requirement for these bytes.

Remediation

This issue has been acknowledged by Anza Technology, Inc., and a fix was implemented in commit 1777181c ↗.

3.9. AccountInfo raw pointer has an unclear provenance invariant

Target	sdk/pinocchio/src/account_info.rs		
Category	Coding Mistakes	**Severity**	Informational
Likelihood	N/A	**Impact**	Informational

Description

`AccountInfo::data_ptr` returns a pointer to the `[u8]` data stored after the `Account` structure by deriving it through the `AccountInfo` struct's `raw: *mut Account` pointer. For this to be valid, an `AccountInfo` must always be constructed from a pointer whose provenance includes both the `Account` and the following `[u8]` data.

Impact

Since with the current codebase, the entrypoint's `input: *mut u8` that covers all of the `(Account, [u8])` pairs is the original allocation that all of the `AccountInfo::raw` pointers are derived from, this does not currently lead to undefined behavior; however, violating this invariant in the future may lead to the current code invoking undefined behavior.

Recommendations

The most direct way to represent the trailing `[u8]` data would be add a `data: [u8]` field to `Account`. This would make `Account` a dynamically-sized type, which would require using `ptr::to_raw_parts` and `ptr::from_raw_parts_mut` to convert between a wide pointer to an `Account` (which has the size in its metadata) and a thin pointer (with the size stored in the `Account`'s `data_len`).

Alternatively, it can be documented on `AccountInfo` that the `raw` pointer used to construct it must be derived from a larger allocation containing both the `Account` and the data.

Remediation

This issue has been acknowledged by Anza Technology, Inc., and a fix was implemented in PR #254 ↗.

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. New off-chain implementation of `impl_sysvar_get` may return different `ProgramError`

The previous implementation of getting various system variables did not differentiate between on chain and off chain and would always return the constant `UNSUPPORTED_SYSVAR` if the sysvar was unsupported.

In the new implementation, there is a difference in the on-chain and off-chain code, where the off-chain portion may return a value that does not match `UNSUPPORTED_SYSVAR`.

```
macro_rules! impl_sysvar_get {
    ($syscall_name:ident) => {
        fn get() -> Result<Self, $crate::program_error::ProgramError> {
            let mut var = core::mem::MaybeUninit::<Self>::uninit();
            let var_addr = var.as_mut_ptr() as *mut _ as *mut u8;

            #[cfg(target_os = "solana")]
            let result = unsafe { $crate::syscalls::$syscall_name(var_addr) };

            #[cfg(not(target_os = "solana"))]
            let result = core::hint::black_box(var_addr as *const _ as u64);

            match result {
                // SAFETY: The syscall initialized the memory.
                $crate::SUCCESS => Ok(unsafe { var.assume_init() }),
                e => Err(e.into()),
            }
        }
    };
}
```

If current programs rely on the return value of these functions to be `UNSUPPORTED_SYSVAR`, they might break in the case an unexpected value is returned. We would suggest keeping the behaviour consistent with the old program, by always returning the `UNSUPPORTED_SYSVAR` where appropriate.

This issue has been acknowledged by Anza Technology, Inc., and a fix was implemented in PR #181 ↗.

4.2. Usage of magic numbers in various locations

It is generally not recommended to use magic numbers in code, as they make it harder to understand the logic and increase difficulty of refactoring the code.

Some examples of magic numbers were found in account_info.rs,

```
pub fn can_borrow_lamports(&self) -> Result<(), ProgramError> {
    let borrow_state = unsafe { (*self.raw).borrow_state };

    // check if mutable borrow is already taken
    if borrow_state & 0b_1000_0000 != 0 {
        return Err(ProgramError::AccountBorrowFailed);
    }

    // check if we have reached the max immutable borrow count
    if borrow_state & 0b_0111_0000 == 0b_0111_0000 {
        return Err(ProgramError::AccountBorrowFailed);
    }

    Ok(())
}
```

as well as instructions.rs,

```
fn from(account: &'a AccountInfo) -> Self {
    Account {
        key: offset(account.raw, 8),
        lamports: offset(account.raw, 72),
        data_len: account.data_len() as u64,
        data: offset(account.raw, 88),
        owner: offset(account.raw, 40),
        // The `rent_epoch` field is not present in the `AccountInfo` struct,
        // since the value occurs after the variable data of the account in
        // the runtime input data.
        rent_epoch: 0,
        is_signer: account.is_signer(),
        is_writable: account.is_writable(),
        executable: account.executable(),
        _account_info: PhantomData::<&'a AccountInfo>,
    }
}
```

and more (not listed for brevity).

To increase readability and make working with the code easier, we recommend finding all instances of magic numbers and replacing them with properly named constants.

This issue has been acknowledged by Anza Technology, Inc., and a fix was implemented in PR #191 ↗.

4.3. The `invoke_signed` function now enforces account order

In its current implementation, the `invoked_signed` function does not care about the order of `AccountInfo` passed to it. The only requirement is that the slice of `AccountInfo` contains all that is required for the call to succeed.

Pinocchio's stated goal is to be a near drop-in replacement for the current program, so this change in logic is a small but potentially important difference in behavior.

The new implementation of `invoke_signed` does require the `AccountInfo` to be in the correct order, otherwise the CPI call will fail.

```
pub fn invoke_signed<const ACCOUNTS: usize>(
    instruction: &Instruction,
    account_infos: &[&AccountInfo; ACCOUNTS],
    signers_seeds: &[Signer],
) -> ProgramResult {
    if instruction.accounts.len() < ACCOUNTS {
        return Err(ProgramError::NotEnoughAccountKeys);
    }

    const UNINIT: MaybeUninit<Account> = MaybeUninit::<Account>::uninit();
    let mut accounts = [UNINIT; ACCOUNTS];

    for index in 0..ACCOUNTS {
        let account_info = account_infos[index];
        let account_meta = &instruction.accounts[index];

        if account_info.key() != account_meta.pubkey {
            return Err(ProgramError::InvalidArgument);
        }
        ...
    }
    ...
```

This issue has been acknowledged by Anza Technology, Inc..

4.4. Less robust `Pubkey` definition

In the existing `solana-program` (and therefore `solana-pubkey`) crates, the definition for `Pubkey` is a transparent type that wraps a `[u8; 32]`. This reflects the use of a powerful design pattern called the "New Type" pattern. This design pattern is often used to help provide type-level safety and distinction to what would otherwise be an arbitrary set of 32 bytes, clearly delineating an instance of those bytes as representing a `Pubkey`.

The advantages of this design are that it helps prevent logic errors by enabling the compiler to more readily catch misused functions, providing methods on the type that allow further proper state transitions that are specific to operations performed on `Pubkeys` (such as deriving additional program addresses or seed addresses), and that it makes writing extension methods via extension traits easy.

We noticed the redefinition of `Pubkey` to instead be a type alias for `[u8; 32]`, which we feel is a step backward in the design and goes against many principles that the Rust language and community value and recognize. We believe that this definition, while implemented to make it easier for serialization frameworks to work with `Pubkey`, ultimately has more consequences than benefits. We encourage the use of the original, more robust definition.

This issue has been acknowledged by Anza Technology, Inc..

4.5. Statically compute `max_digits` in log macro

The `impl_log_for_unsigned_integer` macro in the log crate is defined as follows:

```
macro_rules! impl_log_for_unsigned_integer {
    ( $type:tt, $max_digits:literal ) => {
        impl Log for $type {
            ...
        }
        ...
    },
    ...
}
```

Here, `$max_digits` is supposed to reflect the number of digits (in base 10) it takes to record the largest representable figure of type `$type`. We can see uses of this macro within the crate like this:

```
impl_log_for_unsigned_integer!(u8, 3);
impl_log_for_unsigned_integer!(u16, 5);
```

```
impl_log_for_unsigned_integer!(u32, 10);
impl_log_for_unsigned_integer!(u64, 20);
impl_log_for_unsigned_integer!(u128, 39);
// Handle the `usize` type.
#[cfg(target_pointer_width = "32")]
impl_log_for_unsigned_integer!(usize, 10);
#[cfg(target_pointer_width = "64")]
impl_log_for_unsigned_integer!(usize, 20);
```

We propose, instead, statically calculating these figures at compile time. It would have no impact on performance, but it would better maintain the correct figures and eliminate the need for the `cfg` attribute. This can be done with the following computation:

```
const MAX_DIGITS: usize = const { $type::MAX.ilog10() as usize + 1 };
```

This issue has been acknowledged by Anza Technology, Inc., and a fix was implemented in PR #188 ↗.

4.6. Message difference in `process_initialize_immutable_owner`

In the previous implementation of `process_initialize_immutable_owner`, a message was emitted to indicate that the user should upgrade to SPL Token 2022.

```
pub fn process_initialize_immutable_owner(accounts: &[AccountInfo]) ->
    ProgramResult {
    let account_info_iter = &mut accounts.iter();
    let token_account_info = next_account_info(account_info_iter)?;
    let account =
    Account::unpack_unchecked(&token_account_info.data.borrow())?;
    if account.is_initialized() {
        return Err(TokenError::AlreadyInUse.into());
    }
    msg!("Please upgrade to SPL Token 2022 for immutable owner support");
    Ok(())
}
```

This message is not emitted in the new implementation; instead, it is just a comment in the code. We do not think there is any security implication in not emitting the message, but we mention it because we noticed the slight difference in behavior.

```
pub fn process_initialize_immutable_owner(accounts: &[AccountInfo]) ->
    ProgramResult {
    let token_account_info =
    accounts.first().ok_or(ProgramError::NotEnoughAccountKeys)?;

    // SAFETY: single immutable borrow to `token_account_info` account data.
    let account = unsafe {
    load_unchecked::<Account>(token_account_info.borrow_data_unchecked())? };

    if account.is_initialized()? {
        return Err(TokenError::AlreadyInUse.into());
    }
    // Please upgrade to SPL Token 2022 for immutable owner support.
    Ok(())
}
```

This issue has been acknowledged by Anza Technology, Inc..

4.7. The `withdraw_excess_lamports` function does not prevent self-transfers

In the `process_transfer` function, which is responsible for transferring tokens from one account to another, there is a self-transfer check. This check compares the source and destination account to see if they are the same account, which would be a self-transfer that will be treated differently to a regular transfer.

Another function that has the ability to move tokens is the `withdraw_excess_lamports` function. This function is missing a check to see if the source and destination account are matching, thus allowing self-transfers. If these self-transfers are undesirable, a similar check to the `process_transfer` one can be added to the `withdraw_excess_lamports` function.

This issue has been acknowledged by Anza Technology, Inc..

4.8. The `Default` trait implementation for `Rent` sysvar differs from `solana-program`

In `pinocchio`, the `Rent` sysvar has a `Default` trait implementation automatically generated via a `derive` macro.

```
#[repr(C)]
#[derive(Clone, Debug, Default)]
```

```rust
pub struct Rent {
    /// Rental rate in lamports per byte-year
    pub lamports_per_byte_year: u64,

    /// Exemption threshold in years
    pub exemption_threshold: f64,

    /// Burn percentage
    pub burn_percent: u8,
}
```

If one were to instantiate the `Rent` sysvar via its default implementation, all fields would be set to their default values — in this case, 0.

However, the `Rent` sysvar's `Default` implementation is defined in `solana-program` (actually `solana-rent`) in the following way:

```rust
pub const DEFAULT_LAMPORTS_PER_BYTE_YEAR: u64 = 1_000_000_000 / 100 * 365 /
    (1024 * 1024);
pub const DEFAULT_EXEMPTION_THRESHOLD: f64 = 2.0;
pub const DEFAULT_BURN_PERCENT: u8 = 50;

impl Default for Rent {
    fn default() -> Self {
        Self {
            lamports_per_byte_year: DEFAULT_LAMPORTS_PER_BYTE_YEAR,
            exemption_threshold: DEFAULT_EXEMPTION_THRESHOLD,
            burn_percent: DEFAULT_BURN_PERCENT,
        }
    }
}
```

These are nonzero figures, meaning that the expression `Rent::default()` differs in behavior between `pinocchio` and the existing crates.

This issue has been acknowledged by Anza Technology, Inc., and a fix was implemented in PR #189 ↗.

4.9. Missing `unsafe` in `validate_owner` function

The `validate_owner` function in p-token carries with it an additional safety requirement necessary to prevent undefined behavior. Because of this additional requirement not being statically checked by the compiler that influences memory safety, the function should be marked `unsafe` to properly

communicate the risk.

```
#[inline(always)]
#[allow(clippy::arithmetic_side_effects)]
fn validate_owner(
    expected_owner: &Pubkey,
    owner_account_info: &AccountInfo,
    signers: &[AccountInfo],
) -> ProgramResult {
    if expected_owner != owner_account_info.key() {
        return Err(TokenError::OwnerMismatch.into());
    }

    if owner_account_info.data_len() == Multisig::LEN
        && owner_account_info.is_owned_by(&TOKEN_PROGRAM_ID)
    {
        // SAFETY: the caller guarantees that there are no mutable borrows of
        // `owner_account_info` account data and the `load` validates that the
        // account is initialized; additionally, `Multisig` accounts are only
        // ever loaded in this function, which means that previous loads will
        // have already failed by the time we get here.
        let multisig = unsafe {
    load::<Multisig>(owner_account_info.borrow_data_unchecked())? };
    ...
```

This issue has been acknowledged by Anza Technology, Inc., and a fix was implemented in PR #66 ↗.

4.10. Missing `mut` in Pinocchio tests

The tests in `sdk/pinocchio/src/account_info.rs` demonstrate uses of `Ref`. They construct a `Ref` object with a `state` field that is immutable. This is then passed to `NonNull` and set on the `Ref` instance. Later, in the `Drop` implementation, the `state` field is mutated, which is undefined behavior.

It does not appear that a `Ref` or `RefMut` can be easily constructed when *using* the library, so this does not raise this issue to the level of a finding, but it is suggested to change the tests to reflect that `state` should be mutable.

This issue has been acknowledged by Anza Technology, Inc., and a fix was implemented in PR #178 ↗.

4.11. Accounting error in batched instructions

During the audit the client surfaced an issue in the batch processing of instructions.

The new implementation of the token program, p-token, added the ability to process instructions in a batch. The Solana runtime provides some safety guarantees involving state changes to accounts, that are only checked after a transaction has been executed but not finalized. If the state of an account differs from before the transaction, to after the transaction, ownership checks are performed that ensure that account data has not been modified by programs that should not have the ability to. During batch processing, this check can be bypassed for certain instructions like `transfer` which lack explicit ownership checks.

```
if self_transfer || amount == 0 {
    // Validates the token accounts owner since we are not writing
    // to these account.
    check_account_owner(source_account_info)?;
    check_account_owner(destination_account_info)?;
} else {
    // Moves the tokens.

    source_account.set_amount(remaining_amount);

    // SAFETY: single mutable borrow to `destination_account_info` account
    data; the
    // account is guaranteed to be initialized and different than
    // `source_account_info`; it was also already validated to be a token
    // account.
    let destination_account = unsafe {

    load_mut_unchecked::<Account>(destination_account_info.borrow_mut_data_unchecked())
    };
    // Note: The amount of a token account is always within the range of the
    // mint supply (`u64`).
    destination_account.set_amount(destination_account.amount() + amount);

    if source_account.is_native() {
        // SAFETY: single mutable borrow to `source_account_info` lamports.
        let source_lamports = unsafe {
    source_account_info.borrow_mut_lamports_unchecked() };
        *source_lamports = source_lamports
            .checked_sub(amount)
            .ok_or(TokenError::Overflow)?;

        // SAFETY: single mutable borrow to `destination_account_info`
    lamports; the
        // account is already validated to be different from
        // `source_account_info`.
```

```
            let destination_lamports =
                unsafe { destination_account_info.borrow_mut_lamports_unchecked()
        };
            *destination_lamports = destination_lamports
                .checked_add(amount)
                .ok_or(TokenError::Overflow)?;
        }
    }
```

Here, explicit ownership checks are only performed for self transfers or if the transfered amount equals 0. The special handling of "native" accounts could be abused by batch processing transfers involving an account that is attacker controlled and not owned by the token program. As long as the final state of the attacker controlled account exactly matches the state before the transaction, the runtime will not verify ownership and lead to accounting errors.

This issue has been acknowledged by Anza Technology, Inc., and a fix was implemented in PR #80 ↗.

5. Assessment Results

During our assessment on the scoped Pinocchio library and p-token program, we discovered eight findings. One critical issue was found. Three were of high impact, one was of medium impact, and three were of low impact.

Anza Technology, Inc.'s goal with Pinocchio was to create an efficient replacement for `solana-program` that can lower CUs across the ecosystem as well as remove dependencies that could serve as an additional supply-chain risk. To accomplish this, the project uses a not-insignificant amount of `unsafe` and other dangerous functionality. This adds significant risk to this project as well as brittleness to the implementation.

Despite fuzzing p-token and other reviews performed on both p-token and Pinocchio, we discovered high and critical findings that could lead to out-of-bound reads, reading from uninitialized memory, and undefined behavior in programs compiled against Pinocchio. Additionally, the `unsafe` API exposed to consumers of these libraries carry potentially challenging prerequisite safety conditions. The use of the `unsafe` functionality by consumers of these libraries could result in the accidental introduction of subtle vulnerabilities.

Due to the extensive use of `unsafe` and types such as `MaybeUninit`, it is also our opinion that while a particular commit could be determined to be safe, this is a weak assurance and is more likely to be broken with further changes to the code. That is to say, it is relatively trivial to (re)introduce undefined behavior or other vulnerabilities with code that leverages `unsafe` and these other such dangerous features, especially when one considers the frequency of their use within this codebase.

To help combat these problems, we recommend Anza Technology, Inc. incorporate the use of Miri in their CI process to help catch undefined behavior earlier. We also recommend reaudits for any nontrivial changes to these libraries and to significantly expand their test suite to exercise all code paths. This final recommendation will have the additional effect of aiding Miri in the discovery of undefined behavior, catching issues earlier.

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.