Zellic

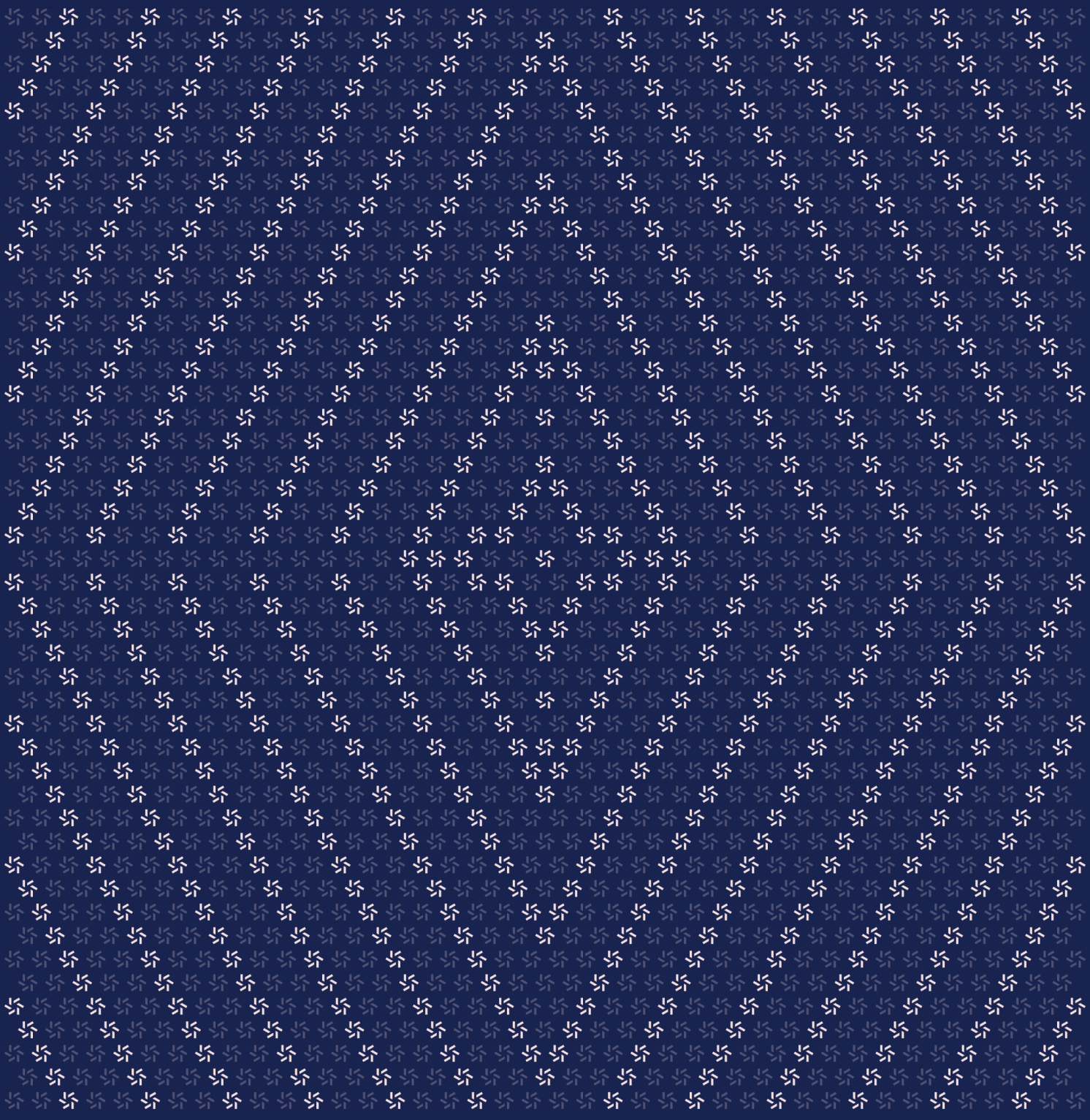
Prepared for
Grug Capital
Warlock Labs

Prepared by
Katerina Belotskaia
Minsun Kim
Zellic

October 15, 2024

SolBLS

Smart Contract Security Assessment



Contents

About Zellic

Zellic is a vulnerability research firm with deep expertise in blockchain security. We specialize in EVM, Move (Aptos and Sui), and Solana as well as Cairo, NEAR, and Cosmos. We review L1s and L2s, cross-chain protocols, wallets and applied cryptography, zero-knowledge circuits, web applications, and more.

Prior to Zellic, we founded the #1 CTF (competitive hacking) team ↗ worldwide in 2020, 2021, and 2023. Our engineers bring a rich set of skills and backgrounds, including cryptography, web security, mobile security, low-level exploitation, and finance. Our background in traditional information security and competitive hacking has enabled us to consistently discover hidden vulnerabilities and develop novel security research, earning us the reputation as the go-to security firm for teams whose rate of innovation outpaces the existing security landscape.

For more on Zellic's ongoing security research initiatives, check out our website zellic.io ↗ and follow @zellic_io ↗ on Twitter. If you are interested in partnering with Zellic, contact us at hello@zellic.io ↗.

1. Overview

1.1. Executive Summary

Zellic conducted a security assessment for Warlock Labs from September 24th to October 1st, 2024. During this engagement, Zellic reviewed SolBLS'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:

- Could the lack of explicit subgroup membership checks potentially introduce vulnerabilities in certain scenarios?
- Does the library's reliance on precompiles for key validation have implications for gas costs and security?

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.

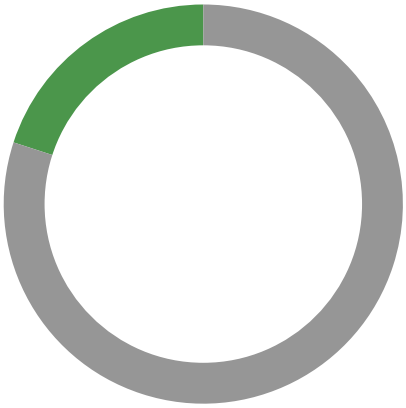
1.4. Results

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

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

Breakdown of Finding Impacts

Impact Level	Count
Critical	0
High	0
Medium	0
Low	1
Informational	4



2. Introduction

2.1. About SolBLS

Warlock Labs contributed the following description of SolBLS:

> A Solidity library for efficient BLS signature verification over the BN254 curve, optimized for on-chain verification.

2.2. Methodology

During a security assessment, Zellic works through standard phases of security auditing, including both automated testing and manual review. These processes can vary significantly per engagement, but the majority of the time is spent on a thorough manual review of the entire scope.

Alongside a variety of tools and analyzers used on an as-needed basis, Zellic focuses primarily on the following classes of security and reliability issues:

Basic coding mistakes. Many critical vulnerabilities in the past have been caused by simple, surface-level mistakes that could have easily been caught ahead of time by code review. Depending on the engagement, we may also employ sophisticated analyzers such as model checkers, theorem provers, fuzzers, and so on as necessary. We also perform a cursory review of the code to familiarize ourselves with the contracts.

Business logic errors. Business logic is the heart of any smart contract application. We examine the specifications and designs for inconsistencies, flaws, and weaknesses that create opportunities for abuse. For example, these include problems like unrealistic tokenomics or dangerous arbitrage opportunities. To the best of our abilities, time permitting, we also review the contract logic to ensure that the code implements the expected functionality as specified in the platform's design documents.

Integration risks. Several well-known exploits have not been the result of any bug within the contract itself; rather, they are an unintended consequence of the contract's interaction with the broader DeFi ecosystem. Time permitting, we review external interactions and summarize the associated risks: for example, flash loan attacks, oracle price manipulation, MEV/sandwich attacks, and so on.

Code maturity. We look for potential improvements in the codebase in general. We look for violations of industry best practices and guidelines and code quality standards. We also provide suggestions for possible optimizations, such as gas optimization, upgradability weaknesses, centralization risks, and so on.

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

Informational.

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

Finally, Zellic provides a list of miscellaneous observations that do not have security impact or are not directly related to the scoped contracts itself. These observations — found in the Discussion (4. ↗) section of the document — may include suggestions for improving the codebase, or general recommendations, but do not necessarily convey that we suggest a code change.

2.3. Scope

The engagement involved a review of the following targets:

SolBLS Contracts

Type	Solidity
Platform	EVM-compatible
Target	solbls
Repository	https://github.com/warlock-labs/solbls ↗
Version	`1ac7407a38df580d3a678aa4b9701667c387ef04`
Programs	`BLS` `ModExp`

2.4. Project Overview

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

Contact Information

The following project manager was associated with the engagement:

Chad McDonald
Engagement Manager
chad@zellic.io ↗

The following consultants were engaged to conduct the assessment:

Katerina Belotskaia
Engineer
kate@zellic.io ↗

Minsun Kim
Engineer
minsun@zellic.io ↗

2.5. Project Timeline

The key dates of the engagement are detailed below.

September 24, 2024	Start of primary review period
September 26, 2024	Kick-off call
October 1, 2024	End of primary review period

3. Detailed Findings

3.1. Invalid curve points accepted by `isValidPublicKey`

Target	BLS		
Category	Coding Mistakes	**Severity**	High
Likelihood	Low	**Impact**	Low

Description

The BLS library implements BLS signatures over the BN254 curve. The BN254 curve comes with a pairing $e : \mathbb{G}_1 \times \mathbb{G}_2 \to \mathbb{G}_T$, with all three abelian groups $\mathbb{G}_1, \mathbb{G}_2, \mathbb{G}_T$ of prime order r. The group $\mathbb{G}_1$ is defined as the subgroup of order r of $E\left(\mathbb{F}_N\right)$, with E the BN254 elliptic curve defined over $\mathbb{F}_N$, and $\mathbb{G}_2$ similarly the subgroup of order r of $E'\left(\mathbb{F}_{N^2}\right)$ for another elliptic curve E' (a sextic twist of E defined over $\mathbb{F}_{N^2}$).

A valid public key is a nonzero point of $\mathbb{G}_2$.[1]

In the BLS library, the `isValidPublicKey` function is passed four $\mathbb{F}_N$ and returns whether these coordinates form a valid public key. It is implemented as follows:

```
/// @notice Check if a given public key is valid (i.e., on the curve)
/// @param publicKey The public key to check
/// @return True if the public key is valid
function isValidPublicKey(uint256[4] memory publicKey)
    internal pure returns (bool) {
    if ((publicKey[0] >= N) || (publicKey[1] >= N) || (publicKey[2] >= N ||
    (publicKey[3] >= N))) {
        return false;
    } else {
        return isOnCurveG2(publicKey);
    }
}
```

This function checks that the four coordinates in $\mathbb{F}_N$ are represented by nonnegative integers smaller than N, to avoid multiple representations for the same public key. The remaining checks are handled by the `isOnCurveG2` function. This function should thus check the following:

1. The coordinates satisfy the affine curve equation. As the zero point of the elliptic curve in affine representation is the point at infinity (i.e., not a point satisfying the curve equation),

[1] Compare for example https://www.ietf.org/archive/id/draft-irtf-cfrg-bls-signature-05.html#name-keyvalidate, section 2.5.

this check will imply that the coordinates represent a nonzero point on the curve.

2. The point lies in the subgroup $\mathbb{G}_2$ (i.e., is of order r).

However, `isOnCurveG2` only carries out the first check but does not verify whether the point lies in the subgroup $\mathbb{G}_2$.

Whether this is a bug in `isOnCurveG2` itself or whether this function is working as intended, with `isValidPublicKey` missing, the check is ambiguous. The function is documented as follows:

```
/// @notice Check if a point is on the G2 curve
/// @param point The point to check
/// @return isOnCurve True if the point is on the curve
function isOnCurveG2(uint256[4] memory point)
    internal pure returns (bool isOnCurve) {
```

The name of the function and description could be interpreted in one of two ways:

1. The function checks whether the coordinates represent a point on the curve that $\mathbb{G}_2$ is a subgroup of.
2. The function checks whether the coordinates represent a point that lies in the subgroup $\mathbb{G}_2$.

This ambiguity can lead to incorrect usage, should a caller assume that this function implemented the second description, even though it implements the first.

Impact

For arguments that represent a nonzero point in $E'\left(\mathbb{F}_{N^2}\right)$ but which are not elements of $\mathbb{G}_2$, the function `isValidPublicKey` will return `true` even though such a point is not a valid public key and should be rejected.

The precise impact of this depends on usage by the user of the BLS library.

Note that `verifySingle`, the function used to verify a signature, does not rely on `isValidPublicKey` and in fact will return `false` for `callSuccess` should the public key not be an element of $\mathbb{G}_2$.

Recommendations

We recommend to change `isValidPublicKey` to also check for subgroup membership. If `isOnCurveG2` is not intended to check subgroup membership, then we recommend to document this clearly, and perhaps also consider renaming the function. In that case, another function could be implemented that carries out the subgroup membership check, which can then be called by `isValidPublicKey`. If instead `isOnCurveG2` is intended to check for subgroup membership, then this

check should be added. In this case, we still recommend to also document more clearly what `isOnCurveG2` does.

Carrying out the subgroup membership check can be done using the ecPairing(BN254 ate-pairing) precompile as follows, which also checks that the coordinates represent a point on the curve:

```
function isElementOfG2(uint256[4] memory pkey) internal view returns (bool) {
    uint256[6] memory input = [
        0, 0,
        pkey[1], pkey[0], pkey[3], pkey[2]
    ];

    bool callSuccess;
    uint256[1] memory out;

    assembly {
        callSuccess := staticcall(sub(gas(), 2000), 8, input, 192, out, 0x20)
    }
    return (callSuccess && (out[0] == 1));
}
```

The function calculates the ate-pairing for an infinity point on `G1` and the provided public key. If a valid `G2` element was provided, `staticcall` would succeed and return `1` as an output; it would fail otherwise. It consumes about 79K gas per call. For a discussion about how ecPairing precompiles check subgroup membership, see Discussion point 4.2. ↗.

The above implementation could replace the implementation of `isOnCurveG2`, or alternatively this could be added as a new function to be called by `isValidPublicKey` instead of `isOnCurveG2`.

Remediation

This issue has been acknowledged by Warlock Labs, and a fix was implemented in commit 5ea30c7a ↗.

Their official response is reproduced below:

> We created the `isElementOfG2` function that is used in `isValidPubKey` to perform both curve membership and subgroup membership checks.

3.2. Behavior of ModexpInverse and ModexpSqrt incompletely documented

Target	ModExp		
Category	Code Maturity	**Severity**	Informational
Likelihood	N/A	**Impact**	Informational

Description

The two libraries ModexpInverse and ModexpSqrt calculate $x^{N-2} \mod N$ as well as $x^{\frac{N+1}{4}} \mod N$ respectively with an addition chain. Here, N is order of the prime field over which the BN254 elliptic curve is defined. For verification that the addchain correctly computes these powers, see Discussion point 4.1. ↗.

The two libraries are documented as follows:

```
/**
 * @title Compute Inverse by Modular Exponentiation
 * @notice Compute $input^(N - 2) mod N$ using Addition Chain method.
 * Where N = 0x30644e72e131a029b85045b68181585d97816a916871ca8d3c208c16d87cfd47
 * and N - 2 =
    0x30644e72e131a029b85045b68181585d97816a916871ca8d3c208c16d87cfd45
 * @dev the function body is generated with the modified addchain script
 * see
    https://github.com/kobigurk/addchain/commit/2c37a2ace567a9bdc680b4e929c94
 * aaaa3ec700f
 */
library ModexpInverse {
    //...
}

/**
 * @title Compute Square Root by Modular Exponentiation
 * @notice Compute $input^{(N + 1) / 4} mod N$ using Addition Chain method.
 * Where N = 0x30644e72e131a029b85045b68181585d97816a916871ca8d3c208c16d87cfd47
 * and (N + 1) / 4 =
    0xc19139cb84c680a6e14116da060561765e05aa45a1c72a34f082305b61f3f52
 */
library ModexpSqrt {
    //...
}
```

The intention of `ModexpInverse.run(x)` and `ModexpSqrt.run(x)` is thus to return the inverse and square root of x in $\mathbb{F}_N$. In the following, we thus discuss how $x^{N-2} \mod N$ and $x^{\frac{N+1}{4}} \mod N$ relate to the inverse and square root of x.

ModexpInverse

- For $x \neq 0, (x^{N-2} \mod N) = (x^{-1} \mod N)$ holds by Fermat's little theorem, so modular inverse is correctly computed.
- For $x = 0$, the library would return `0` despite x's modular inverse not existing. An alternative reasonable behavior would be to revert.

The behavior of the function returning `0` when called for `0` conforms to the specification of `inv0` from RFC 9380 ↗, which makes sense in the context of its use in the BLS library. However, it should be documented that this is the intended behavior.

ModexpSqrt

- Let $s = x^{\frac{N+1}{4}} \mod N$, which is the return value. Then, $(s^2 \mod N) = (x^{\frac{N+1}{2}} \mod N) = (x \times x^{\frac{N-1}{2}} \mod N)$.
- For $x = 0$, $s = 0, (s^2 \mod N) = x = 0$, so the modular square root is correctly computed.
- For nonzero value x, if x's Legendre symbol is equal to 1 (equivalently, if x is a nonzero square), then $(s^2 \mod N) = x$. In this case, modular square root is correctly computed.
- For nonzero value x, if x's Legendre symbol is equal to -1 (equivalently, if x is not a square), then $(s^2 \mod N) = (-x \mod N)$. In this case, s is not a modular square root of x, though x also does not have any square root. Instead, s is a square root of $-x$. An alternative reasonable behavior would be to revert.

Hence, this library returns incorrect values that are not a square root of the input about half the time. [2] Thus, a caller will in general need to check whether the return value is a valid square root. This is what the wrapper function `sqrt` of BLS library does, which saves whether the square-root calculation succeeded in a second return value `hasRoot` with `hasRoot = mulmod(x, x, N) == xx`.

We also note that in contrast to some other functions in the BLS library, both `ModexpInverse.run` and `ModexpSqrt.run` accept arguments that are bigger than or equal to N and will reduce them modulo N.

Impact

The behavior of the functions we discussed above are not cryptographic vulnerabilities themselves, but they can lead to problems when used incorrectly.

In particular, the square root existing for only $\frac{N+1}{2}$ values out of N when N is a prime number may be less known to some users, so it may be possible that some users do not realize the need to check

[2] To be precise, $\frac{N-1}{2}$ of the N possible inputs do not have a square root.

the return value based on the current documentation.

Recommendations

We recommend to fully document the two functions' behavior so that callers do not make any incorrect assumptions. This could be done by adding the points mentioned above to the `@notice`. Concretely, these are the following:

- For ModexpInverse, behavior on input `0`
- For ModexpSqrt, behavior for values without modular square roots
- The fact that not every value has a modular square root modulo N
- That both accept values bigger than or equal to N and will reduce such inputs modulo N

Since the additional validity check always has to be made when using the ModexpSqrt library, one could additionally consider to add a second `hasRoot` return value that indicates whether the first return value is a valid square root of the input. This would make `ModexpSqrt.run` behave similarly to the current `sqrt` wrapper in BLS.sol. The following pseudocode demonstrates what we mean, though see also Finding 3.3. ↗:

```
library ModexpSqrt {
    function run(uint256 t6) internal pure returns (uint256 t0) {
    function run(uint256 t6) internal pure returns (uint256 t0, bool hasRoot)
        {
        // slither-disable-next-line assembly
        assembly {
            let n
    := 0x30644e72e131a029b85045b68181585d97816a916871ca8d3c208c16d87cfd47

            t0 := mulmod(t6, t6, n)
            let t4 := mulmod(t0, t6, n)

            //...

            t0 := mulmod(t0, t1, n)
            t0 := mulmod(t0, t0, n)

            let s2 := mulmod(t0, t0, n)
        }
        hasRoot = (s2 == (t6 % n));
    }
}
```

Remediation

This issue has been acknowledged by Warlock Labs, and a fix was implemented in commit 5ea30c7a ↗.

3.3. Ambiguous behavior of `sqrt` function

Target	BLS		
Category	Code Maturity	**Severity**	Informational
Likelihood	N/A	**Impact**	Informational

Description

The `sqrt` function of the BLS library is implemented as follows:

```
/// @notice Compute the square root of a field element
/// @param xx The element to compute the square root of
/// @return x The square root
/// @return hasRoot True if the square root exists
function sqrt(uint256 xx) internal pure returns (uint256 x, bool hasRoot) {
    x = ModexpSqrt.run(xx);
    hasRoot = mulmod(x, x, N) == xx;
}
```

This function acts as a wrapper around `ModexpSqrt.run`, handling the case of inputs xx that are not squares modulo N, for which `ModexpSqrt.run` still returns a value, which, however, will not be a square root of xx; see Finding 3.2. ↗. It does so by setting `hasRoot` to true if and only if the first return value x is a square root of xx modulo N.

As described in Finding 3.2. ↗, the function `ModexpSqrt.run` allows arguments that are bigger than or equal to N, and it still returns the modular square root of the argument modulo N, if a square root exits.

However, `sqrt` will always return `hasRoot = false` if $xx \geq N$, because `mulmod(x, x, N)` will always be reduced modulo N.

For an argument xx, a return value with `hasRoot = false` can thus mean that $xx \mod N$ does not have a square root modulo N but could also happen when such a square exists, but $xx \mod N$.

Behavior of `sqrt` for arguments bigger than or equal to N is currently not documented.

Impact

Users `sqrt` have no way to distinguish these two cases (no square root existing modulo N and the argument being $\geq N$). Furthermore, as behavior for inputs $xx \geq N$ is not documented, users might not anticipate that the returned `hasRoot` may be false even if $xx \mod N$ has a square root modulo N.

Recommendations

We recommend to document behavior of `sqrt` for arguments $xx \geq N$.

Additionally, it may be reasonable to make `sqrt` revert on such inputs, thereby distinguishing the error case of out-of-range inputs from normal `hasRoot = false` cases on inputs that do not have a modular square root.

Alternatively, the implementation of `sqrt` could be changed to allow arguments $xx \geq N$ as well, by reducing xx modulo N when checking whether x is a square root:

```
function sqrt(uint256 xx) internal pure returns (uint256 x, bool hasRoot) {
    x = ModexpSqrt.run(xx);
    hasRoot = mulmod(x, x, N) == xx;
    hasRoot = mulmod(x, x, N) == (xx % N);
}
```

Remediation

This issue has been acknowledged by Warlock Labs, and a fix was implemented in commit `5ea30c7a` ↗.

Their official response is reproduced below:

> We implemented the suggested check in `sqrt` of BLS.sol that `hasRoot = mulmod(x, x, N) ^= (xx % N)` to handle the case of the base being larger than the modulus, such that `hasRoot` is indicative only of the presence of a valid square root. This behaviour is now documented in both the `@notice` of `ModExp/ModexpSqrt` and in `BLS/sqrt`.

3.4. Addchain method not as efficient as precompile call

Target	ModExp		
Category	Optimization	**Severity**	Informational
Likelihood	N/A	**Impact**	Informational

Description

In audits/internal_audit_july_2024.md, it is said that calling the Ethereum modexp precompile at `0x05` consumes around 14k gas. While the addchain method consumes around 7k gas per call, for functions `inverse` and `sqrt`, it is better to use the addchain compared to the precompile.

> One of the most expensive steps in hash-and-pray is the sqrt step, which takes 14k gas alone by calling the modexp precompile. This is why all of these versions have ported a copy of the ModExp.sol library, which reduces the gas used to about 7k gas. The current version using SvdW is constant-time, and reasonably cheap, at the cost of being difficult to implement.

However, the 14k gas cost for the modexp precompile is outdated. This precompile was first specified in EIP-198, in which the gas costs were set consistent with the 14k number. But later, EIP-2565 reduced the gas cost. The gas cost of exponentiation modulo N as used in the BLS library can be calculated with the following Python script:

```
#!/usr/bin/env python3

import math

def calculate_multiplication_complexity(base_length, modulus_length):
    max_length = max(base_length, modulus_length)
    words = math.ceil(max_length / 8)
    return words**2

def calculate_iteration_count(exponent_length, exponent):
    iteration_count = 0
    if exponent_length <= 32 and exponent == 0: iteration_count = 0
    elif exponent_length <= 32: iteration_count = exponent.bit_length() - 1
    elif exponent_length > 32: iteration_count = (8 * (exponent_length - 32))
    + ((exponent & (2**256 - 1)).bit_length() - 1)
    return max(iteration_count, 1)

def calculate_gas_cost(base_length, modulus_length, exponent_length,
    exponent):
```

```
    multiplication_complexity
    = calculate_multiplication_complexity(base_length, modulus_length)
    iteration_count = calculate_iteration_count(exponent_length, exponent)
    return max(200, math.floor(multiplication_complexity * iteration_count
    / 3))

n = 0x30644e72e131a029b85045b68181585d97816a916871ca8d3c208c16d87cfd47
print(calculate_gas_cost(32, 32, 32, n - 2))
```

Concretely, the above script calculates the gas cost for exponentiation of 256-bit numbers to the power $N-2$ (as used by `inverse`) modulo N. The gas cost returned is 1349. For the exponent $\frac{N+1}{4}$, the gas cost is calculated as 1338. These gas costs fit with `go-ethereum` as well, where the gas cost is implemented at the function `func (c *bigModExp) RequiredGas(input []byte) uint64` in core/vm/contracts.go.

The new, lower gas costs have been active in Ethereum Mainnet since April 15th, 2021. Thus, we suggest updating the two functions in ModExp.sol to use the modexp precompile instead of an addchain, as this would reduce gas usage by more than a factor of three.

We have implemented `inverse` and `sqrt` ourselves using the modexp precompile call like the following:

```
// SPDX-License-Identifier: MIT
pragma solidity ^0.8;

library ModexpInverse {
    function run(uint256 base) internal view returns (uint256 result) {
        bool success;

        assembly {
            let memPtr := mload(0x40)

            mstore(memPtr, 0x20)
            mstore(add(memPtr, 0x20), 0x20)
            mstore(add(memPtr, 0x40), 0x20)

            mstore(add(memPtr, 0x60), base)
            mstore(add(memPtr, 0x80),
    0x30644e72e131a029b85045b68181585d97816a916871ca8d3c208c16d87cfd45)
            mstore(add(memPtr, 0xa0),
    0x30644e72e131a029b85045b68181585d97816a916871ca8d3c208c16d87cfd47)

            success := staticcall(sub(gas(), 2000), 0x05, memPtr, 0xc0,
    memPtr, 0x20)
            result := mload(memPtr)
        }
    }
```

```
}

library ModexpSqrt {
    function run(uint256 base) internal view returns (uint256 result) {
        bool success;

        assembly {
            let memPtr := mload(0x40)

            mstore(memPtr, 0x20)
            mstore(add(memPtr, 0x20), 0x20)
            mstore(add(memPtr, 0x40), 0x20)

            mstore(add(memPtr, 0x60), base)
            mstore(add(memPtr, 0x80),
    0xc19139cb84c680a6e14116da060561765e05aa45a1c72a34f082305b61f3f52)
            mstore(add(memPtr, 0xa0),
    0x30644e72e131a029b85045b68181585d97816a916871ca8d3c208c16d87cfd47)

            success := staticcall(sub(gas(), 2000), 0x05, memPtr, 0xc0,
    memPtr, 0x20)
            result := mload(memPtr)
        }
    }
}
```

We also made a test for calculating 20 samples of `inverse` or `sqrt` functions, and the result is the following:

```
Ran 4 tests for test/modexp.t.sol:CounterTest
[PASS] test_inv1() (gas: 141109)
[PASS] test_inv2() (gas: 37468)
[PASS] test_sqrt1() (gas: 139812)
[PASS] test_sqrt2() (gas: 37227)
Suite result: ok. 4 passed; 0 failed; 0 skipped; finished in 4.86ms (5.21ms
    CPU time)
```

Here, `test_inv1` and `test_sqrt1` are the original implementations, and `test_inv2` and `test_sqrt2` are our implementations using the precompile. According to these results, the on-chain calculations with addchain use around 7k per call, which matches the description quoted initially. However, our version using the precompile call uses less than 2k gas per call instead of 14k. This matches expectations from EIP-2565 and the `go-ethereum` code.

Impact

The functions used to calculate the modular inverse and square root are inefficient in the current implementation, with alternative implementations using the precompile reducing gas by more than 70%.

Recommendations

We recommend to replace the current implementation of the ModexpInverse and ModexpSqrt libraries with the versions that use the modexp precompile that we listed above.

Remediation

This issue has been acknowledged by Warlock Labs, and a fix was implemented in commit 5ea30c7a ↗.

Their official response is reproduced below:

> The core of the ModexpSqrt library (as well as ModexpInv) now relies on the precompile at 0x05 to perform the relevant exponentiation.

3.5. Function `mapToPoint` implemented with inefficient gas usage

Target	BLS		
Category	Optimization	**Severity**	Informational
Likelihood	N/A	**Impact**	Informational

Description

In the library BLS, the function `mapToPoint` is implemented as follows:

```
/// @notice Map field element to E using SvdW
/// @param u Field element to map
/// @return p Point on curve
function mapToPoint(uint256 u) internal view returns (uint256[2] memory p) {
    if (u >= N) revert InvalidFieldElement(u);

    uint256 tv1 = mulmod(mulmod(u, u, N), C1, N);
    uint256 tv2 = addmod(1, tv1, N);
    tv1 = addmod(1, N - tv1, N);
    uint256 tv3 = inverse(mulmod(tv1, tv2, N));
    uint256 tv5 = mulmod(mulmod(mulmod(u, tv1, N), tv3, N), C3, N);
    uint256 x1 = addmod(C2, N - tv5, N);
    uint256 x2 = addmod(C2, tv5, N);
    uint256 tv7 = mulmod(tv2, tv2, N);
    uint256 tv8 = mulmod(tv7, tv3, N);
    uint256 x3 = addmod(Z, mulmod(C4, mulmod(tv8, tv8, N), N), N);

    bool hasRoot;
    uint256 gx;
    if (legendre(g(x1)) == 1) {
        p[0] = x1;
        gx = g(x1);
        (p[1], hasRoot) = sqrt(gx);
        if (!hasRoot) revert MapToPointFailed(gx);
    } else if (legendre(g(x2)) == 1) {
        p[0] = x2;
        gx = g(x2);
        (p[1], hasRoot) = sqrt(gx);
        if (!hasRoot) revert MapToPointFailed(gx);
    } else {
        p[0] = x3;
        gx = g(x3);
```

```
        (p[1], hasRoot) = sqrt(gx);
        if (!hasRoot) revert MapToPointFailed(gx);
    }
    if (sgn0(u) != sgn0(p[1])) {
        p[1] = N - p[1];
    }
}
```

This function implements Shallue-van de Woestijne encoding, which is a method generating a `G1` element from a value in the prime field. This algorithm finishes in a constant time in the worst case, unlike general algorithms like `try-and-increment / hash-and-pray`, whose runtime does not have a theoretical upper bound. However, it is inefficiently implemented regarding the usage of specific functions that consume a lot of gas.

Gas-usage comparison

There are three exponentiation functions used in `mapToPoint`, which are `legendre`, `inverse`, and `sqrt`. They are either implemented with an addchain algorithm (`inverse` and `sqrt`) or with the Ethereum modexp precompile at address `0x05` (`legendre`). According to the description, one modexp precompile call consumes around 14k gas, and both addchain functions `inverse`, `sqrt` consume around 7k gas. The gas cost for the precompile is not correct since 2021 on the Ethereum Mainnet however, with actual gas costs being below 2k; see Finding 3.4. ↗. Thus, a `legendre` call actually costs about 2k gas. Gas usage in `mapToPoint` outside of calls to `legendre`, `inverse`, and `sqrt` are relatively negligible. We will compare how many average calls are made and how much gas it consumes respectively.

We will first summarize how `mapToPoint` works.

- Call `inverse` once — `x1`, `x2`, `x3` are all calculated, and it is mathematically proved at least one of the three x-coordinates will have a corresponding `G1` point.
- Call `legendre` for `g(x1)` and check if it is valid.
    - If it is valid, call `sqrt` for `g(x1)` (about $\frac{1}{2}$ probability).
    - If it is invalid, call `legendre` for `g(x2)` and check if that is valid. If that is valid, call `sqrt` for `g(x2)` (about $\frac{1}{4}$ probability). If that is invalid, call `sqrt` for `g(x3)` (about $\frac{1}{4}$ probability).

Note that `g(x3)`'s square root always exists if `g(x1)` and `g(x2)` are both not squares.

Here are all called exponentiation functions for each case:

- $\frac{1}{2}$ probability: `inverse, legendre, sqrt`
- $\frac{1}{4}$ probability: `inverse, legendre, legendre, sqrt`
- $\frac{1}{4}$ probability: `inverse, legendre, legendre, sqrt`

The best case has three exponentiation function calls, the worst case four, and the average 3.5.

Here is the estimated gas usage with `inverse` and `sqrt`, 7k, and `legendre`, 2k:

- $\frac{1}{2}$ probability: 7k + 2k + 7k
- $\frac{1}{4}$ probability: 7k + 2k + 2k + 7k
- $\frac{1}{4}$ probability: 7k + 2k + 2k + 7k

Thus, the best case costs around 16k gas, the worst case around 18k gas, and the average 17k.

Replacing the implementations of `inverse` and `sqrt` with implementations using the precompile as discussed in Finding 3.4. ↗ will reduce the cost of these functions to about 2k gas as well. With that change, the cost for `mapToPoint` would become

- $\frac{1}{2}$ probability: 3*2k
- $\frac{1}{4}$ probability: 4*2k
- $\frac{1}{4}$ probability: 4*2k

Thus, the best case costs around 6k gas, the worst case around 8k gas, and the average 7k. This is a reduction in average gas costs for `mapToPoint` by 59%.

However, the gas costs can be reduced further by removing the calls to `legendre`, as `sqrt` already returns whether its input was a square or not as a second return value. Concretely, an optimized version of `mapToPoint` could look as follows. Only the part after calculating `x1, x2, x3` is changed.

```
function mapToPoint(uint256 u) internal view returns (uint256[2] memory p) {

    // ..

    bool hasRoot;

    p[0] = x1;
    (p[1], hasRoot) = sqrt(g(p[0]));

    if (!hasRoot) {
        p[0] = x2;
        (p[1], hasRoot) = sqrt(g(p[0]));

        if (!hasRoot) {
            p[0] = x3;
            (p[1], hasRoot) = sqrt(g(p[0]));
        }
    }
}
```

This is the flow for the newer version:

- Call `inverse` once — `x1, x2, x3` are all calculated.
- Call `sqrt` for `g(x1)`, and check if it is valid (about $\frac{1}{2}$ probability). If it is invalid, call `sqrt` for `g(x2)`, and check if it is valid (about $\frac{1}{4}$ probability). If it is invalid, call `sqrt` for `g(x3)` (about $\frac{1}{4}$ probability).

Here are all called exponentiation functions for each case:

- $\frac{1}{2}$ probability: `inverse, sqrt`
- $\frac{1}{4}$ probability: `inverse, sqrt, sqrt`
- $\frac{1}{4}$ probability: `inverse, sqrt, sqrt, sqrt`

It has best case two exponentiation function calls, worst case four, and average 2.75.

Here is the estimated gas usage with `inverse` and `sqrt` both costing 2k gas:

- $\frac{1}{2}$ probability: 2*2k
- $\frac{1}{4}$ probability: 3*2k
- $\frac{1}{4}$ probability: 4*2k

It uses best case 4k gas, worst case 8k gas, and average 5.5k.

The number of exponentiation calls is reduced from 3.5 to 2.75, which is 21% improvement. And the gas usage is reduced from 7k to 5.5k in the average case, which is a further 21% improvement.

Making both changes reduces the average gas cost of `mapToPoint` from 17k to 5.5k, a reduction by 68%.

Impact

Although the function does not have a cryptographical vulnerability, the optimized version reduces the gas usage by 68% when combined with Finding 3.4. ↗. The reduction due to removal of `legendre` calls is 21%. This difference is significant since the calculation is done on chain.

Recommendations

We recommend using the optimized version below:

```
bool hasRoot;

p[0] = x1;
(p[1], hasRoot) = sqrt(g(p[0]));

if (!hasRoot) {
    p[0] = x2;
    (p[1], hasRoot) = sqrt(g(p[0]));

    if (!hasRoot) {
        p[0] = x3;
        (p[1], hasRoot) = sqrt(g(p[0]));
    }
}
```

If this change is made, the functions `expModLegendre` and `legendre` will not be used anymore, so they might be removed if they are not called anywhere else.

Remediation

This issue has been acknowledged by Warlock Labs, and a fix was implemented in commit `5ea30c7a` ↗.

Their official response is reproduced below:

> The SvdW encoding now uses a more efficient implementation of the final steps of the encoding, allowing for the entire deletion of the `legendre` and `modExpLegendre` functions.

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. Checking exponents for ModExp.sol libraries

The two functions in ModExp.sol calculate specific modular exponentiations using long addchains, with implementation of each function consisting of about 300 lines of the following form:

```
// ...
        t0 := mulmod(t0, t5, n)
        t0 := mulmod(t0, t0, n)
        t0 := mulmod(t0, t0, n)
        t0 := mulmod(t0, t0, n)
        t0 := mulmod(t0, t2, n)
// ...
```

To test whether the functions were properly calculating the exponentiation with the correct exponent, we used the following Python code.

```
inv_text = """
            let n :=
    0x30644e72e131a029b85045b68181585d97816a916871ca8d3c208c16d87cfd47
            t0 := mulmod(t2, t2, n)
            let t5 := mulmod(t0, t2, n)

            ...

            t0 := mulmod(t0, t0, n)
            t0 := mulmod(t0, t1, n)
"""

sqrt_text = """
            let n :=
    0x30644e72e131a029b85045b68181585d97816a916871ca8d3c208c16d87cfd47

            t0 := mulmod(t6, t6, n)
            let t4 := mulmod(t0, t6, n)

            ...

            t0 := mulmod(t0, t1, n)
            t0 := mulmod(t0, t0, n)
"""
```

```
n = 0x30644e72e131a029b85045b68181585d97816a916871ca8d3c208c16d87cfd47

def mulmod(a, b, c):
    assert c == n
    return a + b

check_inv, check_sqrt = True, True

if check_inv:
    t2 = 1

    text = inv_text.split("\n")

    for cmd in text:
        if len(cmd) == 0:
            continue
        cmd = cmd.replace("let", "").replace(":=", "=")

        while cmd[0] == " " or cmd[0] == "\t":
            cmd = cmd[1:]

        if cmd[0] == "n":
            continue

        exec(cmd)

    assert t0 == n - 2

if check_sqrt:
    t6 = 1

    text = sqrt_text.split("\n")

    for cmd in text:
        if len(cmd) == 0:
            continue
        cmd = cmd.replace("let", "").replace(":=", "=")

        while cmd[0] == " " or cmd[0] == "\t":
            cmd = cmd[1:]

        if cmd[0] == "n":
            continue

        exec(cmd)

    assert t0 == (n + 1) // 4
```

The strings `inv_text` and `sqrt_text`, which are shortened in the above snippet, were directly pasted from the assembly section of `run` functions of the two libraries.

Both of the libraries only consist of `mulmod` instructions with modulus N. We set the input variable's exponent as 1 and made `mulmod` act as addition, thereby tracking the exponents. The final return values match the required exponent ($N - 2$ and $\frac{N+1}{4}$, respectively).

4.2. The 0x08 ecPairing precompile call internal implementation

The ecPairing precompile at address `0x08` takes multiple couples of one `G1` point and `G2` point as an input.

Let us say the elliptic curves `G1` and `G2` are on `E1` and `E2`, respectively.

Because every point on `E1` is a `G1` element, `G1` is the same group as `E1`. So to check if a point is on `E1`, it is good enough to check if it is a `G1` element. However, `G2` is a subgroup of order `q = 218...617`, while `E2` is an elliptic curve with order `10069 * 5864401 * 1875725156269 * 19762036451288124722871705034201332756068320190696890 9 * q`. So a proper check for whether some point is a `G2` element should be done in two steps.

1. Check if the point is on `E2`.
2. Check if the point's scalar multiplication by `q` is equal to the point at infinity of `E2`.

It is clearly stated that the `G2` subgroup check is properly made in EIP-197 ↗.

> For `G_2`, in addition to that, the order of the element has to be checked to be equal to the group order `q = 218...617`.

We went through how the ecPairing call is implemented with go-ethereum ↗'s most common implementation to see if it actually satisfies the specification.

The ecPairing call fails for invalid `G2` points through the following calls:

- `runBn256Pairing` in core/vm/contracts.go
- at `newTwistPoint` in core/vm/contracts.go
- at `G2.Unmarshal` in crypto/bn256/cloudflare/bn256.go
- at `twistPoint.IsOnCurve` in crypto/bn256/cloudflare/twist.go

Function `twistPoint.IsOnCurve` is implemented as the following:

```
// IsOnCurve returns true iff c is on the curve.
func (c *twistPoint) IsOnCurve() bool {
    c.MakeAffine()
    if c.IsInfinity() {
        return true
```

```
    }

    y2, x3 := &gfP2{}, &gfP2{}
    y2.Square(&c.y)
    x3.Square(&c.x).Mul(x3, &c.x).Add(x3, twistB)

    if *y2 != *x3 {
        return false
    }
    cneg := &twistPoint{}
    cneg.Mul(c, Order)
    return cneg.z.IsZero()
}
```

Here, it requires `G2 * q = E2(0)` by `cneg.Mul(c, Order)`, `return cneg.z.IsZero()`, so it filters points that are on `E2`, but the order is not `q`. It would instantly raise an error if `twistPoint.IsOnCurve` returns false, which makes the ecPairing precompile call fail, which satisfies the specification.

4.3. Inconsistent `verifySingle` return type

In BLS.sol, the function `verifySingle`'s return type is two boolean values where the second value is whether the `staticcall` succeeded.

```
function verifySingle(uint256[2] memory signature, uint256[4] memory pubkey,
    uint256[2] memory message)
    internal
    view
    returns (bool pairingSuccess, bool callSuccess)
{
    ...
    assembly {
        callSuccess := staticcall(sub(gas(), 2000), 8, input, 384, out, 0x20)
    }
    return (out[0] != 0, callSuccess);
}
```

The `verifySingle` function returns the result of a low-level call instead of checking it and reverting execution in case of an unsuccessful call, as the `hashToPoint` function does. This inconsistency could lead to potential issues when using this library, for example, if the function's result is ignored and assumed to be verified within `verifySingle`.

In README.md, there exists an example usage, which is identical to the following:

```
contract MyContract {
    using BLS for *;
```

```
    function verifySignature(
        uint256[2] memory signature,
        uint256[4] memory pubkey,
        uint256[2] memory message
    ) public view returns (bool) {
        // First, check if the signature and public key are valid
        require(BLS.isValidSignature(signature), "Invalid signature");
        require(BLS.isValidPublicKey(pubkey), "Invalid public key");

        // Hash the message to a point on the curve
        uint256[2] memory hashedMessage = BLS.hashToPoint("domain",
    abi.encodePacked(message));

        // Verify the signature
        return BLS.verifySingle(signature, pubkey, hashedMessage);
    }
}
```

We can notice that MyContract was written under the assumption that `verifySingle` returns a single boolean value, and it skipped the result of the low-level call. For the function matching the codebase's consistency, we recommend one of the following:

1. Add `if (!callSuccess) revert ...` and return one boolean value `out[0] != 0`.
2. Return one boolean value `callSuccess && out[0] != 0`.

Remediation

This issue has been acknowledged by Warlock Labs, and a fix was implemented in commit 5ea30c7a ↗.

Their official response is reproduced below:

> This function now returns a single boolean indicating both success in the static call of the precompile and of the result of execution of the precompile.

5. Assessment Results

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

During our assessment on the scoped SolBLS contracts, we discovered five findings. No critical issues were found. One finding was of low impact and the other 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.