Biconomy Secp256r1

Smart Contract Security Assessment

October 30, 2023

Prepared for:

Sachin Tomar, Chirag, Aman

Biconomy Labs

Prepared by:

Malte Leip and Mohit Sharma

Zellic Inc.

Contents

About Zellic

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

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

1 Executive Summary

Zellic conducted a security assessment for Biconomy Labs from October 23rd to October 25th, 2023. During this engagement, Zellic reviewed Biconomy Secp256r1's code for security vulnerabilities, design issues, and general weaknesses in security posture.

1.1 Goals of the Assessment

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

- Does `Secp256r1` implement ECDSA signature verification on the secp256r1 curve correctly?
- Why does `test case ID 345: extreme value for k and s^-1` from Project Wycheproof fail?
- Why does `test case ID 285: k*G has a large x-coordinate` from Project Wycheproof fail?
- Do the functions `_jAdd` and `_modifiedJacobianDouble`, when passed valid Jacobian coordinates for points on the secp256r1 curve, always correctly return Jacobian coordinates for their sum and double, respectively?

1.2 Non-goals and Limitations

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

- Impact of bugs on users of the Secp256r1 library, except through use of the `Verify` function.

We considered potential security issues when using `Verify` that were caused by bugs or nonadherence to specifications for ECDSA signature validation, but we did not consider how the Secp256r1 library is used in practice.

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

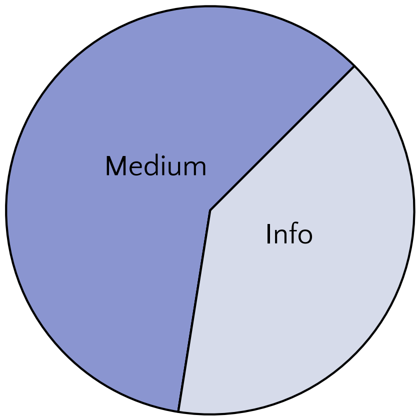
1.3 Results

During our assessment on the scoped Biconomy Secp256r1 contracts, we discovered five findings. No critical issues were found. Three findings were of medium impact and the remaining findings were informational in nature.

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

Breakdown of Finding Impacts

Impact Level	Count
Critical	0
High	0
Medium	3
Low	0
Informational	2



2 Introduction

2.1 About Biconomy Secp256r1

Biconomy Secp256r1 implements ECDSA signature verification on the secp256r1 curve.

2.2 Methodology

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

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

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

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

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

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

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

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

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

2.3 Scope

The engagement involved a review of the following targets:

Biconomy Secp256r1 Contracts

Repository	https://github.com/bcnmy/scw-contracts/
Version	scw-contracts: `674e49a92c59e38e3ddbbd91deb008f7f64a4ed3`
Program	Secp256r1
Type	Solidity
Platform	EVM-compatible

2.4 Project Overview

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

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:

Malte Leip, Engineer
malte@zellic.io

Mohit Sharma, Engineer
mohit@zellic.io

2.5 Project Timeline

The key dates of the engagement are detailed below.

October 23, 2023	Start of primary review period
October 25, 2023	End of primary review period

3 Detailed Findings

3.1 Function _jAdd returns an incorrect result if the summands are equal

- **Target**: Secp256r1
- **Category**: Coding Mistakes
- **Likelihood**: Low
- **Severity**: High
- **Impact**: Medium

Description

The `_jAdd(uint p1, uint p2, uint p3, uint q1, uint q2, uint q3)` function's purpose is to calculate the sum of the two points `(p1:p2:p3)` and `(q1:q2:q3)` on the elliptic curve secp256r1. Both summands as well as the result are represented with Jacobian coordinates.

However, an issue arises when the represented points are equal: the function returns the incorrect result `(0,0,0)`.

Impact

The value returned to an (indirect) caller of `_jAdd` will be invalid if the two summands passed to `_jAdd` represent equal points, with an impact depending on how the return value is used. Our analysis focused on the impact on the Secp256r1 library's `Verify` function, which is assumed to be the only function called by users of this library.

We came to the conclusion (detailed reasoning can be found in section 4.2) that an attacker could only make use of this bug through `Verify` in order to produce

1. Valid signatures for a key whose private key they know, which, however, get erroneously rejected by `Verify`
2. Invalid signatures for a key whose private key they know, which, however, get erroneously accepted by `Verify`

We thus believe that this bug does *not* help an attacker craft signatures that pass verification even though the attacker was not able to produce a legitimate valid signature.

That a properly generated valid signature triggers this bug is extremely unlikely.

The impact on the security of projects making use of the Secp256r1 library for signature verification is highly dependent on how signatures are otherwise used. See section 4.1 for a discussion of this as well as the reason for our severity rating.

This bug is the root cause of the failure of `test case ID 345: extreme value for k and s^-1` from Project Wycheproof.

Recommendations

Consider adding a check for equality of the two summands in `_jAdd` and, in case of equality, calculate the result using `_modifiedJacobianDouble` instead, as that function implements the specialized formulas intended for precisely this case.

When checking whether the points represented by Jacobian coordinates `(p1:p2:p3)` and `(q1:q2:q3)` are equal, note that $(x\colon y\colon z)$ and $(a^2x\colon a^3x\colon az)$ represent the same point if a is nonzero. One way to check equality is then to convert both points to affine coordinates, with a check like this:

```
(px, py) = _affineFromJacobian(p1, p2, p3);
(qx, qy) = _affineFromJacobian(q1, q2, q3);
if ((px == qx) && (py == qy)) {
    return _modifiedJacobianDouble(p1, p2, p3);
}
```

Alternatively, given that the code in `_jAdd` already calculates values `u1`, `u2`, `s1`, and `s2` as in the following display, we can also check equality by checking whether `(u1 == u2) && (s1 == s2)` after having previously handled the case `(p3 == 0) || (q3 == 0)`.[1]

```
let
    pd
:= 0xFFFFFFFF00000001000000000000000000000000FFFFFFFFFFFFFFFFFFFFFFFF
let z1z1 := mulmod(p3, p3, pd) // Z1Z1 = Z1^2
let z2z2 := mulmod(q3, q3, pd) // Z2Z2 = Z2^2

let u1 := mulmod(p1, z2z2, pd) // U1 = X1*Z2Z2
let u2 := mulmod(q1, z1z1, pd) // U2 = X2*Z1Z1
```

[1] We argue why this check is correct. By converting to affine coordinates we see that points $(x : y : z)$ and $(x' : y' : z')$ are equal if and only if either $z = z' = 0$, or z and z' are nonzero and $xz^{-2} = x'z'^{-2}$ and $yz^{-3} = y'z'^{-3}$. We are assuming that $z \neq 0$ and $z' \neq 0$, so then these points are equal if and only if $xz^{-2} = x'z'^{-2}$ and $yz^{-3} = y'z'^{-3}$, which is equivalent to $xz'^2 = x'z^2$ and $yz'^3 = y'z^3$.

```
let s1 := mulmod(p2, mulmod(z2z2, q3, pd), pd) // S1 = Y1*Z2*Z2Z2
let s2 := mulmod(q2, mulmod(z1z1, p3, pd), pd) // S2 = Y2*Z1*Z1Z1
```

Given that this bug cannot be used by an attacker to pass signature validation for a forged signature unless they could also have created a valid signature for it, it is also an option to not remediate this bug. In that case, it should be carefully considered whether an attacker could use the scenarios above maliciously when taking into account interaction with other components. See 4.1 for a discussion of this risk.

Remediation

This issue has been acknowledged by Biconomy Labs, and fixes were implemented in the following commits:

- 983b699d
- f7e03db2

3.2 Valid signatures with large value for `r` are rejected

- **Target**: Secp256r1
- **Category**: Coding Mistakes
- **Likelihood**: Low
- **Severity**: High
- **Impact**: Medium

Description

The `VerifyWithPrecompute` function handles verification of ECDSA signatures. Signatures consist of a pair `(r, s)` of integers $0 < r, s < n$, where n is the order of the elliptic curve secp256r1. The signature is ultimately accepted if and only if the x-coordinate `x` of a computed point on the elliptic curve satisfies `x == r`, as can be seen in the code snippet below.

```
(x, y) = ShamirMultJacobian(points, u1, u2);
return (x == r);
```

However, the elliptic curve secp256r1 is defined over the finite field $\mathbb{F}_p$, so the x-coordinate `x` will be an element of $\mathbb{F}_p$ and be represented by an integer satisfying $0 \leq x < p$. Specifications[2] state that a signature should be accepted if `x % n == r`. As $n < p$, it can happen that `x % n == r` but `x` $\neq$ `r`, so some signatures that should be accepted are not.

Impact

That a properly generated valid signature will hit this bug by accident is extremly unlikely (it will happen roughly once every `10^39` signatures). The Project Wycheproof test vector shows, however, that it is possible to generate such signatures on purpose. The impact on the security of projects making use of the Secp256r1 library for signature verification is highly dependent on how signatures are otherwise used. See section 4.1 for a discussion of this as well as the reason for our severity rating.

This bug is the root cause of the failure of `test case ID 285: k*G has a large x-coordinate` from Project Wycheproof.

Recommendations

Replace `return (x == r);` by `return ((x % nn) == r);`.

[2] See, for example, section 6.4.2 of https://nvlpubs.nist.gov/nistpubs/FIPS/NIST.FIPS.186-5.pdf.

Remediation

This issue has been acknowledged by Biconomy Labs, and a fix was implemented in commit 983b699d.

3.3 Validity of public keys is not checked

- **Target**: Secp256r1
- **Category**: Coding Mistakes
- **Likelihood**: Low
- **Severity**: High
- **Impact**: Medium

Description

The `Verify` function does not check the validity of the public key `passKey`. To be valid, a public key needs to [3]

1. not be the point at infinity,
2. have coordinates (x, y) satisfying $0 \leq x, y < p$, and
3. satisfy the equation $y^2 = x^3 + ax + b$ modulo `p`.

The public key is only used after conversion to Jacobian coordinates in `_preComputeJacobianPoints` with `JPoint(passKey.pubKeyX, passKey.pubKeyY, 1)`, which is never the point at infinity. The `_affineFromJacobian` function uses the convention that `(0,0)` in affine coordinates represents the point at infinity. So for this special case, conversion as `JPoint(passKey.pubKeyX, passKey.pubKeyY, 1)` would be incorrect. But given that the point at infinity is not a valid public key anyway, this is not an issue if instead the public key `(0,0)` is rejected by recognizing that `(0,0)` does not lie on the curve.

As x and y coordinates of `passKey` always get reduced modulo `p` in calculations, the missing check for property 2 means that `Verify` will in effect check the signature for the public key with coordinates `(x % p, y % p)`. This means that for some public keys `(x,y)`, where $x < 2^{256} - p$ or $y < 2^{256} - p$, there exists another pair `(x', y')` — for example, `(x+p, y)` — that can be used as a public key and for which signatures made for `(x,y)` would also verify.

Finally, if the public key passed to `Verify` does not lie on the curve, then results returned by `Verify` do not have a meaningful interpretation.

Impact

Whether the possibility an attacker could generate two different keys for which the same signature is valid is a problem depends on how the caller uses public keys and signature verification.

[3] See for example the recommendation in NIST SP 800-186, Appendix D.1.1.

This bug allows an attacker to generate public keys together with signatures that will be rejected by verification algorithms that validate the public key but will be accepted by `Verify`. The impact on the security of projects making use of the Secp256r1 library for signature verification is highly dependent on how signatures are otherwise used. See section 4.1 for a discussion of this as well as the reason for our severity rating.

Recommendations

Ensure that `(passKey.pubKeyX, passKey.pubKeyY)` is a valid public key for the secp256r1 curve. One option is to check this in the `Verify` function. If this is instead ensured by callers to `Verify`, then one could alternatively document that `Verify` assumes validity of the public key and that the caller must ensure this.

Remediation

This issue has been acknowledged by Biconomy Labs, and fixes were implemented in the following commits:

- f7e03db2
- 55d6e09c

3.4 Invalid Jacobian coordinates used for the point at infinity

- **Target**: Secp256r1
- **Category**: Coding Mistakes
- **Likelihood**: N/A
- **Severity**: Informational
- **Impact**: Informational

Description

The functions `ShamirMultJacobian` and `_preComputeJacobianPoints` use `(0, 0, 0)` with the intention to represent the point at infinity in Jacobian coordinates. However, this is not a valid point in Jacobian coordinates. The point at infinity is represented in Jacobian coordinates with `(c^2, c^3, 0)`, with `0 < c < p` and exponentiation done modulo `p` [4].

Impact

As `_affineFromJacobian` and `_jAdd` check for an argument being the point at infinity by only comparing the last component with 0, they work as intended anyway. The function `_modifiedJacobianDouble` will return `(0,0,0)` if passed `(0,0,0)`. Results are thus currently correct if `(0, 0, 0)` is treated as an alias for the point at infinity.

Recommendations

Consider changing `(0,0,0)` to `(1,1,0)` in the two places; or, if it is preferred to keep `(0,0,0)` as an efficiency trick to save gas, document that this is intentional and that functions such as `_jAdd`, `_modifiedJacobianDouble`, and `_affineFromJacobian` must treat `(0,0,0)` as the point at infinity. In the latter case, we recommend adding test cases for this as well.

Remediation

This issue has been acknowledged by Biconomy Labs, and fixes were implemented in the following commits:

- f7e03db2
- 55d6e09c
- 43525074

4 `p` refers to the prime over which the elliptic curve is defined.

3.5 Test coverage

- **Target**: Secp256r1
- **Category**: Code Maturity
- **Likelihood**: N/A
- **Severity**: Informational
- **Impact**: Informational

Description

Only the `Verify` function is tested using the Wycheproof test vectors. It could be beneficial to also test other functions directly with unit tests.

Impact

Bugs such as that in section 3.1 could have been found without the Wycheproof ECDSA test vectors, and the root cause could have been found more easily by testing `_jAdd` directly for common edge cases of elliptic curve addition.

Recommendations

Consider adding tests for other functions as well. Specifically for elliptic curve addition `P + Q`, edge cases to test for would be

- `P=0`
- `Q=0`
- `P=Q`
- `P=-Q`

Note that if, for example, `P=(px:py:pz)` and `Q=(qx:qy:qz)` in Jacobian coordinates, then `P=Q` is equivalent to `px = c^2 * qx`, `py = c^3 * qy`, and `pz = c * qz`, for some `0 < c < pp`, with all equalities being modulo `pp`. Tests should thus also include cases `P=Q` where `c` is different than 1 (and similarly for the other three cases).

Remediation

This issue has been acknowledged by Biconomy Labs.

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 Possible impact of signature verification deviating from specifications

Findings 3.1, 3.2, and 3.3 imply that an attacker can generate valid signatures for a key they control, which, however, are erroneously rejected by `Verify` (3.1 and 3.2) or generate invalid signatures for a key they control, which are erroneously accepted (3.1 and 3.3). The impact of this deviation is dependent on how signatures are otherwise used in the entire system that uses the Secp256r1 library that we audited.

The main point of possible concern would be if the same signature is independently verified in two (or more) places, with different code. The system as a whole might be built on the assumption that a signature is either valid or invalid and that all signature verifications will agree on validity. Bugs like the three referenced above then imply that an attacker could craft a signature that breaks this assumption. Consequences of this in concrete cases would be highly dependent on the specific scenario and require careful reasoning to analyze the interplay between all involved components.

When considering a cryptographic library for signature verification such as Secp256r1, we thus consider signatures being accepted if and only if they are valid according to their specification[5] to be the central guarantee the library makes to users. We thus consider any deviations that can be triggered in practice to be bugs of high severity.

4.2 Hypothetical adversaries that can trigger the addition bug through `Verify` can already recover the private key from the public key alone

The following claim is intended as a security reduction argument. Security of ECDSA over secp256r1 as a cryptographic scheme relies on the assumption that, given only a public key, it is computationally intractable to compute the private key from it. We

5 Intentional deviations from the relevant standard for the cryptographic system should be clearly documented.

assume that this is true. Then the below claim argues that if an attacker that only has the public key could use this to efficiently produce `(r, s, e)` that triggers the bug described in Finding 3.1, then this attacker would also be able to efficiently recover the private key from the public key alone. But this would contradict the assumption that this is computationally intractable. Hence, it must be in practice impossible for an attacker to produce such `(r, s, e)` if they only knew the public key.

Claim: An adversary, who knows arguments `(passKey, r, s, e)` that passed to `Verify` will cause a call to `_jAdd` in which the two summands are equal, can use them to efficiently compute the private key corresponding to the public key `passKey`.

Proof: The following diagram depicts the call graph between `Verify` and `_jAdd`.

$$\begin{tikzcd}
 & \boxed{\texttt{\_jAdd}} & \\
\boxed{\texttt{\_jPointAdd}} \arrow[ur] & & \boxed{\texttt{ShamirMultJacobian}} \arrow[ul] \\
\boxed{\texttt{\_preComputeJacobianPoints}} \arrow[u] & & \boxed{\texttt{VerifyWithPrecompute}} \arrow[u] \\
 & \boxed{\texttt{Verify}} \arrow[ul] \arrow[ur] &
\end{tikzcd}$$

We argue that along both branches, triggering the bug implies an efficiently computable, nontrivial linear relation in the elliptic curve between the generator G and the public key, which we shall call Q. This means there are elements α and β of $\mathbb{F}_n$, where n is the (prime) order of the elliptic curve, such that $\alpha \cdot G + \beta \cdot Q = 0$ and $(\alpha, \beta) \neq (0, 0)$. Should we have $Q = 0$, then the corresponding private key is $d = 0$, so we assume $Q \neq 0$ from now on. Then we must have $\beta \neq 0$ so we can define $d = -\alpha\beta^{-1}$, which will then satisfy $d \cdot G = Q$, so d is the private key corresponding to Q.

Let us now consider the left part of the call graph. The function `Verify` calls `_preComputeJacobianPoints`, passing the public key `passKey` as the only argument. The function `_preComputeJacobianPoints` in turn calculates an array `points` of points on the elliptic curve represented in Jacobian coordinates[6], with `points[4i+j]` = $i \cdot G + j \cdot Q$, for $0 \leq i, j < 4$.

6 The point at infinity 0 will though additionally be represented by `(0,0,0)`, see 3.4.

```
points[0] = JPoint(0, 0, 0);
points[1] = JPoint(passKey.pubKeyX, passKey.pubKeyY, 1); // u2
points[2] = _jPointDouble(points[1]);
points[3] = _jPointAdd(points[1], points[2]);

points[4] = JPoint(gx, gy, 1); // u1Points[1]
points[5] = _jPointAdd(points[4], points[1]);
points[6] = _jPointAdd(points[4], points[2]);
points[7] = _jPointAdd(points[4], points[3]);

points[8] = _jPointDouble(points[4]); // u1Points[2]
points[9] = _jPointAdd(points[8], points[1]);
points[10] = _jPointAdd(points[8], points[2]);
points[11] = _jPointAdd(points[8], points[3]);

points[12] = _jPointAdd(points[4], points[8]); // u1Points[3]
points[13] = _jPointAdd(points[12], points[1]);
points[14] = _jPointAdd(points[12], points[2]);
points[15] = _jPointAdd(points[12], points[3]);
```

There are three types of `_jAdd` calls being made through the wrapper `_jPointAdd` when computing this table:[7]

1. One summand is Q; the other is $2 \cdot Q$ (for `points[3]`).
2. One summand is G; the other is $2 \cdot G$ (for `points[12]`).
3. One summand is $\alpha \cdot G$; the other is $\beta \cdot Q$ for $\alpha, \beta \in \{1, 2, 3\}$ (for `points[i]` with $i \in \{5, 6, 7, 9, 10, 11, 13, 14, 15\}$).

In each case, equality of the two summands implies a nontrivial linear relation between G and Q.

Finally, let us consider the right part of the call graph, where we need to look into a call to `ShamirMultJacobian(JPoint[16] memory points, uint u1, uint u2)`, where `points` is precisely the precomputed list of points discussed and where the concrete values of `u1` and `u2` are irrelevant for the following argument. This function uses Shamir's trick

[7] Each entry here is under the assumption that previous calls to `_jAdd` did not trigger the bug yet, which we may.

to efficiently compute $u_1 \cdot G + u_2 \cdot Q$ using the precomputed points. There is only one call to `_jAdd` in this function, namely

```
(x, y, z) = _jAdd(
    x,
    y,
    z,
    points[index].x,
    points[index].y,
    points[index].z
);
```

The arguments `(x,y,z)` to `_jAdd` will represent the point $4\alpha \cdot G + 4\beta \cdot Q$ where α = `(u1 >> (256-2*bits)) & ((1 << (256 - 2*bits)) - 1)` and β = `(u2 >> (256-2*bits)) & ((1 << (256 - 2*bits)) - 1)`. The call to `_jAdd` will only be carried out if `index` is nonzero, and in that case, we know that we will have `points[index]`$= \gamma \cdot G + \delta \cdot Q$ with $0 \leq \gamma, \delta < 4$, not both zero. As the values of `u1` and `u2` are the result of a computation modulo n in `VerifyWithPrecompute`, they must be smaller than n; from which, it follows that we must have $4\alpha, 4\beta < n$. This implies that $(4\alpha - \gamma) \cdot G + (4\beta - \delta) \cdot Q = 0$ is a nontrivial linear relation between G and Q.

This ends the proof of the claim.

As an additional check on this argument, we produced a script[8] that recovers the secret key from the public key and a test vector that triggers this bug.

4.3 Validity of modular subtraction in `_jAdd` and `_modifiedJacobianDouble`

As the EVM does not have a `submod` instruction, the functions `_jAdd` and `_modifiedJacobianDouble` use the following pattern several times.

```
assembly {
    if lt(a, b) {
        a := add(p, a)
    }
    let c := sub(a, b)
```

[8] The script was provided to Biconomy Labs.

```
}
```

The assumption here is that a and b both satisfy $0 \leq a, b < p$, and the intention is to have `c = (a - b) % p`. If $0 \leq b \leq a < p$, then $0 \leq a - b < p$ holds, so plain subtraction can be used. In the case $a < b$, it holds that $0 < (a + p) - b < p$, so $(a + p) - b = (a - b) + p$ is the value needed for c.

We note that while the `add` instruction could overflow (as p is a 256-bit number), as both `add` and `sub` are calculated modulo 2^{256}, the end result for c will still be correct.

4.4 Documentation

While there are several comments in Secp256r1.sol, including NatSpec documentation, some of these are outdated. For example, the NatSpec parameters for `Verify` do not correspond to the actual parameters the function takes, and the description of the `_jAdd` function suggests this function computes a doubling.[9]

Making the naming of variables in the `_jAdd` function (and to an extent, `_modifiedJacobianDouble`) more consistent would make it easier to verify correctness of the calculations. While the parameters are called `p1, p2, p3, q1, q2, q3`, the comments and variable names used in the function suggest `x1, y1, z1, x2, y2, z2`.

The comment before `_modifiedJacobianDouble` suggests this function is operating on modified Jacobian coordinates, but it actually uses the usual Jacobian coordinates. With modified Jacobian coordinates, one would pass four arguments, `(x, y, z, w)` where `w=a*z^4`.

4.5 Possible gas savings in `_jAdd`

The method used by `_jAdd` to calculate the sum of two points given in Jacobian coordinates is to use the formulas termed “add-2007-bl” in the Explicit-Formulas Database. In the EVM, the gas cost of `addmod` and `mulmod` is the same, and there is no cheaper way to square or multiply by a power of 2. This makes the formulas termed “add-1998-cmo-2” more efficient.[10] In the verification of the `_jAdd` formulas we carried out in section 5.1, the change would correspond to removing factors of two, so that at the

9 The linked formula also neither fits `_jAdd` nor `_modifiedJacobianDouble`.

10 The formulas termed “add-2007-bl” use one more squaring as well as several extra additions and multiplications by 2 compared to “add-1998-cmo-2”, while saving a single multiplication. This can be more efficient in environments in which multiplication is much more expensive than these other operations.

end in, for example, case 3, one obtains `(r1,r2,r3)` $= (x_3, y_3, z_3)$ directly rather than `(r1,r2,r3)` $= (4x_3, 8y_3, 2z_3)$ — both represent the same point on the elliptic curve.

Biconomy Labs was provided a proof of concept patch for this optimization, which saved 1.48% gas when running the Wycheproof test suite.[11]

[11] These are savings for the Wycheproof contract as a whole; the savings for the `_jAdd` function itself will thus be larger.

5 Verification of correctness of elliptic curve arithmetic

We manually verified correctness of the crucial `_jAdd` and `_modifiedJacobianDouble` functions. As ground truth for addition on elliptic curves using Jacobian coordinates, we used section 2.6.2 of Washington's textbook on elliptic curves.[12]

5.1 Module: Secp256r1.sol

Secp256r1 constants

The Secp256r1 contract contains constants pertaining to the elliptic curve secp256r1. We checked the constants given against section 2.4.2 in *SEC 2: Recommended Elliptic Curve Domain Parameters*.

Curve constants are correct

- ☑ `gx` is the x-coordinate of the base point.
- ☑ `gy` is the x-coordinate of the base point.
- ☑ `pp` is the order of the finite field over which secp256r1 is defined.
- ☑ `nn` is the order of the curve.
- ☑ `a` and `b` are the parameters such that the Weierstrass equation for secp256r1 is given by $y^2 = x^3 + ax + b$.

Function: `_jAdd(uint256 p1, uint256 p2, uint256 p3, uint256 q1, uint256 q2, uint256 q3)`

This function takes as parameters Jacobian coordinates `(p1:p2:p3)` and `(q1:q2:q3)` of points on the elliptic curve secp256r1 and returns Jacobian coordinates representing their sum. Additionally, if a point is `(0,0,0)`, then the other point is returned (see Finding 3.4).

Inputs

- (`p1`, `p2`, `p3`)

[12] L. C. Washington, *Elliptic curves: number theory and cryptography*, 2nd ed. (Boca Raton, FL: Chapman & Hall/CRC), 2008.

- (q1, q2, q3)

Validation: No checks in this function.

Impact: For each of `(p1,p2,p3)` and `(q1,q2,q3)`, the caller should ensure that these are either valid Jacobian coordinates for a point on the secp256r1 curve or `(0,0,0)`.

Correctness:

- ☑ `pd` agrees with the constant `pp` defined in the library.
- ☑ Assuming `(p1,p2,p3)=(0,0,0)`, the function returns `(q1,q2,q3)`.
- ☑ Assuming `(q1,q2,q3)=(0,0,0)`, the function returns `(p1,p2,p3)`.
- ☐ Assuming `(p1:p2:p3)` and `(q1:q2:q3)` are valid Jacobian coordinates for points on the secp256r1 curve, `(r1:r2:r3) = (p1:p2:p3) + (q1:q2:q3)`.
  - ☐ 1. This holds when `(p1:p2:p3)=(q1:q2:q3)` (i.e., they represent the same point on the curve).
  - ☑ 2. This holds when `(p1:p2:p3)=-(q1:q2:q3)` (i.e., they represent additively inverse points on the curve).
  - ☑ 3. This holds when neither conditions above are the case.
- ☑ The return values satisfy `0 ≤ r1, r2, r3 < pd`, as long as the corresponding property holds for the arguments (for this check we refer also to section 4.3).

Detailed steps taken to check correctness of the result: Assuming `(p1:p2:p3)` and `(q1:q2:q3)` are valid Jacobian coordinates for points on the secp256r1 curve, we compare each of the steps of the computation done in the function with the reference book,[12] case $P_1 \neq \pm P_2$ (though we do not assume this here). Equality is here to be taken modulo `pd`. Here we take `(p1,p2,p3)` and `(q1,q2,q3)` to correspond to (x_1, y_1, z_1) and (x_2, y_2, z_2) in the book.

- ☑ `z1z2` before redefinition in the last lines is what is called z_1^2 in the reference book.
- ☑ `z2z2` is what is called z_2^2 in the reference book.
- ☑ `u1` is what is called r in the reference book.
- ☑ `u2` is what is called s in the reference book.
- ☑ `s1` is what is called t in the reference book.
- ☑ `s2` is what is called u in the reference book.
- ☑ `p3q3` is what is called $z_1 + z_2$ in the reference book.
- ☑ `h` is what is called v in the reference book.
- ☑ `i` is what is called $4v^2$ in the reference book.
- ☑ `j` is what is called $4v^3$ in the reference book.
- ☑ `rr` is what is called $2w$ in the reference book.
- ☑ `r1` after the first assignment is what is called $4w^2$ in the reference book.
- ☑ `v` is what is called $4rv^2$ in the reference book.

- ☑ `j2v` before redefinition in the last lines is what is called $4v^3+8rv^2$ in the reference book.
- ☑ `r1` after the last assignment is what is called $4x_3$ in the reference book.
- ☑ `s12j` is what is called $8tv^3$ in the reference book.
- ☑ `r2` after the first assignment is what is called $8(rv^2-x_3)w$ in the reference book.
- ☑ `r2` after the last assignment is what is called $8y_3$ in the reference book.
- ☑ `z1z1` after the last assignment is what is called $z_1^2+z_2^2$ in the reference book.
- ☑ `j2v` after the last assignment is what is called $z_1^2+2z_1z_2+z_2^2$ in the reference book.
- ☑ `r3` is what is called $2z_3$ in the reference book.

Case 1: Points are equal. In this case, the result is not correct. We will have `u1=u2` and `s1=s2`, which implies that all variables defined afterwards are zero. See Finding 3.1.

Case 2: Points are additive inverses. For easier notation, we use notation from Washington's book.[12] If the points are each other's additive inverse, then there must be a nonzero $c \pmod p$ so that $(x_2,y_2,z_2) = (c^2x_1, -c^3y_1, cz_1)$. Then it follows that $r = c^2c_1z_1^2 = s$ and $t = c^3y_1z_1^3 = -u$, and hence

$$(x_3:y_3:z_3) = (4u^2:-8u^3:0) = \big((-2u)^2\cdot 1 : (-2u)^3\cdot 1 : (-2u)\cdot 0\big) = (1:1:0)$$

Case 3: Points are different and not additive inverses. Then we obtain the point

$$(\texttt{r1:r2:r3}) = (4x_3:8y_3:2z_3) = (2^2x_3:2^3y_3:2z_3) = (x_3:y_3:z_3)$$

which is the correct result according to the book.[12]

Function: `_modifiedJacobianDouble(uint256 x, uint256 y, uint256 z)`

This function takes as parameters Jacobian coordinates `(x:y:z)` of a point on the elliptic curve secp256r1 and returns Jacobian coordinates representing double that point. Additionally, when passed `(0,0,0)`, the function should return `(0,0,0)`. (See Finding 3.4.)

Inputs

- (x, y, z):

Validation: No checks in this function.

Impact: Caller should ensure that `(x:y:z)` are valid Jacobian coordinates for a point on the secp256r1 curve, or `(x,y,z)=(0,0,0)`.

Correctness:

- ☑ pd agrees with the constant pp defined in the library.
- ☑ The inline constant used for a agree with the constant a defined in the library.
- ☑ Assuming `(x:y:z)` are valid Jacobian coordinates for a point on the secp256r1 curve, `(x3:y3:z3) = 2(x:y:z)`.
- ☑ If `(x,y,z)=(0,0,0)`, then the function returns `(0,0,0)`.
- ☑ The return values satisfy `0 ≤ x3, y3, z3 < pd` (for this check, we refer also to section 4.3).

Detailed steps taken to check correctness of the result: Assuming `(x,y,z)` are valid Jacobian coordinates for a point on the secp256r1 curve, we compare each of the steps of the computation done in the function with the reference book.[12] Equality is here to be taken modulo pd.

- ☑ z2 is what is called z_1^2 in the reference book.
- ☑ az4 is what is called az_1^4 in the reference book.
- ☑ y2 is what is called y_1^2 in the reference book.
- ☑ s is what is called v in the reference book.
- ☑ u is what is called $8y_1^4$ in the reference book.
- ☑ m is what is called w in the reference book.
- ☑ twos is what is called $2v$ in the reference book.
- ☑ m2 is what is called w^2 in the reference book.
- ☑ x3 is what is called x_3 in the reference book.
- ☑ After the first assignment, y3 is what is called $(v - x_3)w$ in the reference book.
- ☑ After the last assignment, y3 is what is called y_3 in the reference book.
- ☑ z3 is what is called z_3 in the reference book.

6 Assessment Results

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

During our assessment on the scoped Biconomy Secp256r1 contracts, we discovered five findings. No critical issues were found. Three findings were of medium impact and the remaining findings were informational in nature. Biconomy Labs acknowledged all findings and implemented fixes.

6.1 Disclaimer

This assessment does not provide any warranties about finding all possible issues within its scope; in other words, the evaluation results do not guarantee the absence of any subsequent issues. Zellic, of course, also cannot make guarantees about any code added to the project after the version reviewed during our assessment. Furthermore, because a single assessment can never be considered comprehensive, we always recommend multiple independent assessments paired with a bug bounty program.

For each finding, Zellic provides a recommended solution. All code samples in these recommendations are intended to convey how an issue may be resolved (i.e., the idea), but they may not be tested or functional code. These recommendations are not exhaustive, and we encourage our partners to consider them as a starting point for further discussion. We are happy to provide additional guidance and advice as needed.

Finally, the contents of this assessment report are for informational purposes only; do not construe any information in this report as legal, tax, investment, or financial advice. Nothing contained in this report constitutes a solicitation or endorsement of a project by Zellic.