Veridise. Auditing Report

Hardening Blockchain Security with Formal Methods

FOR

P256Verifier

Veridise Inc.
October 5, 2023

► **Prepared For:**

Daimo
https://daimo.xyz/

► **Prepared By:**

Daniel Domínguez Álvarez
Jacob Van Geffen
Bryan Tan

► **Contact Us:** contact@veridise.com

► **Version History:**

Oct. 6, 2023	V1

Contents

Contents iii

1 Executive Summary 1

2 Project Dashboard 3

3 Audit Goals and Scope 5
- 3.1 Audit Goals . . . 5
- 3.2 Audit Methodology & Scope . . . 5
- 3.3 Classification of Vulnerabilities . . . 6

4 Vulnerability Report 7
- 4.1 Detailed Description of Issues . . . 8
  - 4.1.1 V-P256-VUL-001: Missing check to prevent signature malleability . . . 8
  - 4.1.2 V-P256-VUL-002: Valid public keys of the form (0, y) are rejected . . . 9
  - 4.1.3 V-P256-VUL-003: Potential clarity improvements in P256Verifier . . . 10
  - 4.1.4 V-P256-VUL-004: Undocumented behavior of modInv when called with u=0 . . . 11

Executive Summary 1

From Sep. 13, 2023 to Sep. 26, 2023, Daimo engaged Veridise to review the security of their P256Verifier project, a Solidity smart contract implementation of ECDSA signature verification for the NIST P-256 curve (also known as secp256r1). The smart contract is designed to be a drop-in replacement for the precompiled contract proposed in EIP-7212. Veridise conducted the assessment over 6 person-weeks, with 3 engineers reviewing code over 2 weeks on commit `4887c97`. The security assessment was performed in the same audit as that of the Daimo project. The auditing strategy involved a tool-assisted analysis of the source code performed by Veridise engineers as well as extensive manual auditing.

Code assessment. The P256Verifier developers provided the source code of P256Verifier for review*. The source code appears to be influenced by other ECDSA implementations such as FreshCryptoLib and blst but otherwise seems to be mostly written by the developers. The source code contained some documentation in the form of READMEs and documentation comments on functions and storage variables. The source code also contained a test suite, which the Veridise auditors noted checks the output of the P256Verifier on the secp256r1 test vectors of the wycheproof† project.

Summary of issues detected. The audit uncovered 4 issues, consisting of 1 medium issue, 1 warning, and 2 informational issues. The medium-severity issue involves a missing check for signature malleability (V-P256-VUL-001), the warning identifies a case where two valid public keys may be falsely rejected (V-P256-VUL-002), and the informational issues document parts of the code that the auditors found confusing to read (V-P256-VUL-003, V-P256-VUL-004). The P256Verifier developers resolved all of the reported issues.

Disclaimer. We hope that this report is informative but provide no warranty of any kind, explicit or implied. The contents of this report should not be construed as a complete guarantee that the system is secure in all dimensions. In no event shall Veridise or any of its employees be liable for any claim, damages or other liability, whether in an action of contract, tort or otherwise, arising from, out of or in connection with the results reported here.

* The source code is publicly available at `https://github.com/daimo-eth/p256-verifier`
† `https://github.com/google/wycheproof`

Project Dashboard 2

Table 2.1: Application Summary.

Name	Version	Type	Platform
P256Verifier	`4887c97`	Solidity	Ethereum

Table 2.2: Engagement Summary.

Dates	Method	Consultants Engaged	Level of Effort
Sep. 13 - Sep. 26, 2023	Manual & Tools	3	6 person-weeks

Table 2.3: Vulnerability Summary.

Name	Number	Resolved
Critical-Severity Issues	0	0
High-Severity Issues	0	0
Medium-Severity Issues	1	1
Low-Severity Issues	0	0
Warning-Severity Issues	1	1
Informational-Severity Issues	2	2
TOTAL	4	4

Table 2.4: Category Breakdown.

Name	Number
Data Validation	2
Logic Error	1
Maintainability	1

3 Audit Goals and Scope

3.1 Audit Goals

The engagement was scoped to provide a security assessment of the P256Verifier smart contract. In our audit, we sought to answer questions such as:

- Does the smart contract correctly validate a given ECDSA public key?
- Does the behavior of the implementation match the behavior described in EIP-7212?
- Are there any unstated assumptions that are not clearly documented?
- Are the curve parameters correctly set in the code?
- Is the Strauss-Shamir trick correctly implemented?
- Are elliptic curve operations such as point addition, scalar multiplication, etc. always given valid points on the curve?
- Do the point operations correctly handle cases such as infinity, same point, additive inverse, etc.?

3.2 Audit Methodology & Scope

Audit Methodology. To address the questions above, our audit involved a combination of human experts and automated program analysis & testing tools. In particular, we conducted our audit with the aid of the following techniques:

- *Static analysis.* To identify potential common vulnerabilities, we leveraged our custom smart contract analysis tool Vanguard. Tools such as this are designed to find instances of common smart contract vulnerabilities, such as reentrancy and uninitialized variables.

Scope. The scope of this audit is limited to the `P256Verifier.sol` file of the source code provided by the P256Verifier developers.

Methodology. The Veridise auditors reviewed relevant specifications such as EIP-7212, inspected the provided tests, and read the P256Verifier documentation. They then began a manual audit of the code assisted by static analysis.

References. During the audit, the Veridise auditors compared the implementation to the procedures described in the following documents:

- Standards for Efficient Cryptography 1 (SEC 1), Ver. 2.0. https://www.secg.org/sec1-v2.pdf
- National Institute of Standards and Technology (2023) Digital Signature Standard (DSS). (Department of Commerce, Washington, D.C.), Federal Information Processing Standards Publication (FIPS) NIST FIPS 186-5. https://doi.org/10.6028/NIST.FIPS.186-5

- Chen L, Moody D, Regenscheid A, Robinson A, Randall K (2023) Recommendations for Discrete Logarithm-based Cryptography: Elliptic Curve Domain Parameters. (National Institute of Standards and Technology, Gaithersburg, MD), NIST Special Publication (SP) NIST SP 800-186. `https://doi.org/10.6028/NIST.SP.800-186`

3.3 Classification of Vulnerabilities

When Veridise auditors discover a possible security vulnerability, they must estimate its severity by weighing its potential impact against the likelihood that a problem will arise. Table 3.1 shows how our auditors weigh this information to estimate the severity of a given issue.

Table 3.1: Severity Breakdown.

	Somewhat Bad	Bad	Very Bad	Protocol Breaking
Not Likely	Info	Warning	Low	Medium
Likely	Warning	Low	Medium	High
Very Likely	Low	Medium	High	Critical

In this case, we judge the likelihood of a vulnerability as follows in Table 3.2:

Table 3.2: Likelihood Breakdown

Not Likely	A small set of users must make a specific mistake
Likely	Requires a complex series of steps by almost any user(s) - OR - Requires a small set of users to perform an action
Very Likely	Can be easily performed by almost anyone

In addition, we judge the impact of a vulnerability as follows in Table 3.3:

Table 3.3: Impact Breakdown

Somewhat Bad	Inconveniences a small number of users and can be fixed by the user
Bad	Affects a large number of people and can be fixed by the user - OR - Affects a very small number of people and requires aid to fix
Very Bad	Affects a large number of people and requires aid to fix - OR - Disrupts the intended behavior of the protocol for a small group of users through no fault of their own
Protocol Breaking	Disrupts the intended behavior of the protocol for a large group of users through no fault of their own

Vulnerability Report 4

In this section, we describe the vulnerabilities found during our audit. For each issue found, we log the type of the issue, its severity, location in the code base, and its current status (i.e., acknowledged, fixed, etc.). Table 4.1 summarizes the issues discovered:

Table 4.1: Summary of Discovered Vulnerabilities.

ID	Description	Severity	Status
V-P256-VUL-001	Missing check to prevent signature malleability	Medium	Intended Behavior
V-P256-VUL-002	Valid public keys of the form (0, y) are rejected	Warning	Fixed
V-P256-VUL-003	Potential clarity improvements in P256Verifier	Info	Fixed
V-P256-VUL-004	Undocumented behavior of modInv when called w	Info	Fixed

4.1 Detailed Description of Issues

4.1.1 V-P256-VUL-001: Missing check to prevent signature malleability

Severity	Medium	**Commit**	4887c97
Type	Data Validation	**Status**	Intended Behavior
File(s)	`P256Verifier.sol`		
Location(s)	ecdsa_verify()		
Confirmed Fix At	N/A		

The `ecdsa_verify()` function is the main entry point to ECDSA signature verification implementation in `P256Verifier`. As part of its checks, it validates that the components of the given signature `(r, s)` are both in the range `[1, n - 1]`. However, it does not perform any additional validation on the `s` value.

Although the `P256Verifier` is designed to mimic the behavior of the precompiled contract specified in EIP-7212, it does not include the following check on `s`:

> Verify that s is equal to or less than half of the order of the subgroup to prevent signature malleability.

Impact As noted in Appendix B.3 of SEC 1 Ver 2.0 [PDF], a valid signature `(r, s)` may be transformed into another valid signature `(r, -s mod n)`. Although this is "not regarded as a forgery," omitting the check would allow both signatures to be used. This may affect downstream applications that falsely assume that signatures are unique for a given message, leading to signature malleability attacks.

Recommendation If full compliance with EIP-7212 is desired, the additional check on `s` should be added. Note that this will require signers to only produce signatures satisfying `s <= n / 2`.

Developer Response The developers noted that the contract is designed to prefer to mimic the behavior of the signature verification procedure in SEC 1 and the various NIST documents:

> Good catch.
>
> EIP-7212 is still in draft. We recommended that they remove the malleability check to match the NIST spec exactly: `https://github.com/ethereum/EIPs/pull/7676`
>
> Correspondingly, there's no malleability check in P256Verifier.

4.1.2 V-P256-VUL-002: Valid public keys of the form (0, y) are rejected

Severity	Warning	**Commit**	4887c97
Type	Logic Error	**Status**	Fixed
File(s)	`P256Verifier.sol`		
Location(s)	ecAff_isOnCurve()		
Confirmed Fix At	301328c		

The internal function `ecAff_isOnCurve()` is used to validate whether a public key, given in affine coordinates `(x, y)`, is a valid point on curve P-256. The function validates that all of the following are true:

1. `x` and `y` are both nonzero.
2. `x` and `y` are both strictly less than `p`.

Compared to the "Elliptic Curve Public Key Validation Primitive" procedure described in SEC 1 Ver. 2.0 [PDF], Section 3.2.2.1, the check (1) is slightly too strict and may reject valid public keys with the form `(0, y)` for nonzero `y`. For curve P-256, there exist two valid points with a zero `x` and a nonzero `y`.

```
/**
 * @dev Check if a point in affine coordinates is on the curve
 * Reject 0 point at infinity.
 */
function ecAff_isOnCurve(
    uint256 x,
    uint256 y
) internal pure returns (bool) {
    if (0 == x || x >= p || 0 == y || y >= p) {
        return false;
    }
```

Snippet 4.1: Relevant lines in `ecAff_isOnCurve()`

Impact The two valid public keys of the form `(0, y)` with nonzero `y` may be falsely rejected by the `P256Verifier`:

- $y = 69528327468847610065686496900697922508397251637412376320436699849860351814667$
- $y = 46263761741508638697010950048709651021688891777877937875096931459006746039284$

However, given the large number of points on curve P-256, it is highly unlikely for someone to generate a keypair for which the public key is exactly one of these two points.

Recommendation Change the `0 == x || 0 == y` clauses to instead compare the point with infinity (the fake point `(0, 0)` in the implementation).

Developer Response The developers noted:

> Great catch, thank you. I don't think it impacts Daimo, but it is necessary for our goal of having P256Verifier match the NIST spec (and EIP-7212) exactly. We'll update the check as recommended.

4.1.3 V-P256-VUL-003: Potential clarity improvements in P256Verifier

Severity	Info	**Commit**	4887c97
Type	Maintainability	**Status**	Fixed
File(s)	`P256Verifier.sol`		
Location(s)	See description		
Confirmed Fix At	402e7b4		

There are several places in the `P256Verifier` source code where the clarity of the code could be improved:

- The `ecAff_isOnCurve()` function implements a procedure to validate a public key, similar to the one described in Section 3.2.2.1 of SEC 1 Ver. 2.0 [PDF]. Since this is only called to validate the public key, we recommend renaming this to `ecAff_validatePublicKey()`.
- `ecAff_isOnCurve()` does not include a check that the public key is in the same subgroup as the base point of the curve. For curve P-256, this check can be safely omitted as the property is implied by (1) the order of the base point being equal to the order of the curve; and (2) the check that the public key is a valid point contained in the curve. We recommend adding a comment documenting this fact.
- The `ecAff_isZero()` function is only used in the function `ecAff_add`, where it checks whether the given point is infinity. We recommend renaming the function to `ecAff_isInfinity ()`. Secondly, the documentation comment of `ecAff_isZero()` states "Check if the curve is the zero curve in affine rep". This comment does not seem to use standard terminology; we recommend changing the comment to "Check if the given point is infinity in affine rep".
- `ecAff_isZero()` returns true if and only if `y == 0`. Because infinity is defined as the point `(0, 0)`, it would seem more proper to define the function as returning true if and only if `x == 0 && y == 0`. However, since curve P-256 does not contain any points of the form `(x, 0)` for nonzero `x`, the predicate `y == 0` would imply that the point is infinity (assuming that the point is contained in the curve). We recommend documenting this fact in a comment.
- Several functions such as `ecZZ_SetAff()` and `ecZZ_dadd_affine()` will compare a given point in XYZZ coordinates against infinity by checking whether `zz == 0 && zzz == 0`. To improve code readability, we recommend moving this check into a `ecZZ_isInfinity()` function.
- To check whether the point in affine coordinates is infinity, the `ecZZ_dadd_affine()` function will check `y2 == 0`. To improve readability, we recommend replacing this check with `ecAff_isZero()`.

4.1.4 V-P256-VUL-004: Undocumented behavior of modInv when called with u=0

Severity	Info	**Commit**	4887c97
Type	Data Validation	**Status**	Fixed
File(s)	`P256Verifier.sol`		
Location(s)	modInv()		
Confirmed Fix At	ecf94ce		

The `modInv()` function computes the integer $u^{-1} \mod f$ (where f is a prime number). This is computed by calling the `modexp` precompiled contract. When `u` is 0, the `modInv()` function returns 0. However, the multiplicative inverse of `0` is not defined for integers modulo `f`. For clarity, we recommend documenting this deviation from the mathematical definition of the multiplicative inverse.

```
function modInv(uint256 u, uint256 f, uint256 minus_2modf) internal view returns (
    uint256 result, bool success) {
    bytes memory ret;
    (success, ret) = (address(0x05).staticcall(abi.encode(32, 32, 32, u, minus_2modf,
     f)));
    result = abi.decode(ret, (uint256));
}
```

Snippet 4.2: Implementation of `modInv()`

Note that the only location where `0` can be supplied to the `u` argument of `modInv()` is in `eeZZ_mulmuladd_S_asm()`. However, this can only occur if the final result of the point addition is the point infinity, in which case `X` will correctly be set to 0.

```
uint256 zzInv;
(zzInv, success) = pModInv(zz);
X = mulmod(X, zzInv, p); // X/zz
```

Snippet 4.3: Relevant lines in `eeZZ_mulmuladd_S_asm()`. `zz` can be zero here. Note that `pModInv` is implemented as `modInv(zz, p, p-2)`.

Recommendation Add comments clarifying this behavior.