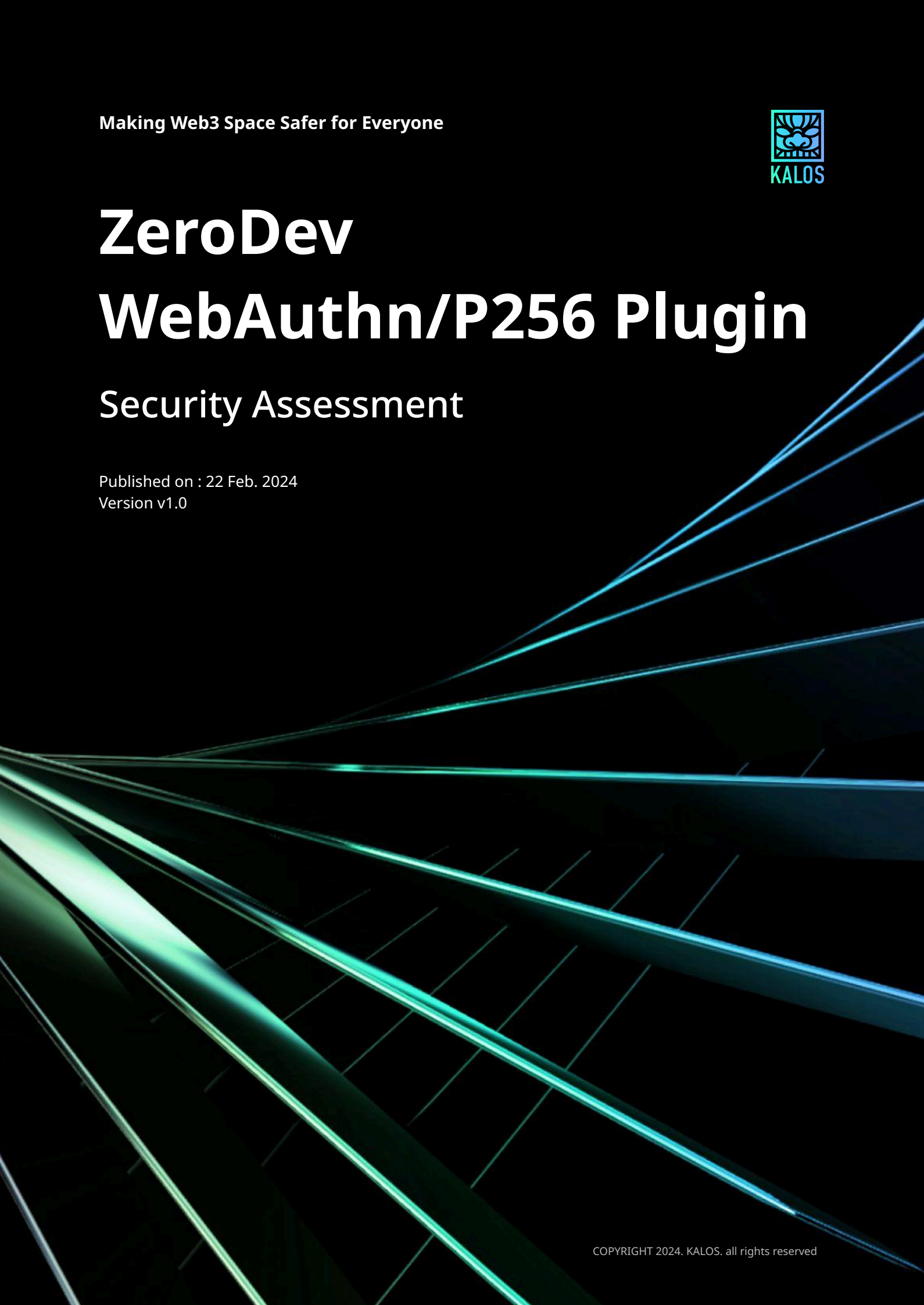
Making Web3 Space Safer for Everyone

KALOS

ZeroDev WebAuthn/P256 Plugin

Security Assessment

Published on : 22 Feb. 2024
Version v1.0



Security Report Published by KALOS

v1.0 22 Feb. 2024

Auditor : Jade Han

hojung han

Found issues

Severity of Issues	Findings	Resolved	Acknowledged	Comment
Critical	-	-	-	-
High	-	-	-	-
Medium	-	-	-	-
Low	2	2	-	-
Tips	-	-	-	-

TABLE OF CONTENTS

ABOUT US

Making Web3 Space Safer for Everyone

Pioneering a safer Web3 space since 2018, KALOS proudly won 2nd place in the Paradigm CTF 2023. As a leader in the global blockchain industry, we unite the finest in Web3 security expertise.

Our team consists of top security researchers with expertise in blockchain/smart contracts and experience in bounty hunting. Specializing in the audit of mainnets, DeFi protocols, bridges, and the zkEVM protocol, KALOS has successfully safeguarded billions in crypto assets.

Supported by grants from the Ethereum Foundation and the Community Fund, we are dedicated to innovating and enhancing Web3 security, ensuring that our clients' digital assets are securely protected in the highly volatile and ever-evolving Web3 landscape.

Inquiries: audit@kalos.xyz
Website: https://kalos.xyz

Executive Summary

Purpose of this report

This report was prepared to audit the security of the project developed by the ZeroDev team. KALOS conducted the audit focusing on whether the system created by the ZeroDev team is soundly implemented and designed as specified in the published materials, in addition to the safety and security of the project.
In detail, we have focused on the following

- Denial of Service
- Access Control of Various Storage Variables
- Access Control of Important Functions
- Freezing of User Assets
- Theft of User Assets
- Unhandled Exceptions
- Compatibility Testing with Bundler

Codebase Submitted for the Audit

The codes used in this Audit can be found on GitHub

- p256 folder in https://github.com/zerodevapp/kernel-plugins

The last commit of the code used for this Audit is

- dfc53f96dc4513d3c410881adb722cc618878d47

The last commit of the patched code is

- fc1c0f15463016bc8e97bc28e4b835728487e53d

Audit Timeline

Date	Event
2024/02/21	Audit Initiation
2024/02/22	Delivery of v1.0 report.

Findings

KALOS found 0 High, 0 medium, and 1 Low severity issues. There are 1 Tips issues explained that would improve the code’s usability or efficiency upon modification.

Severity	Issue	Status
Low	Potential Gas Cost Manipulation via changing usePrecompiled flag	(Resolved)
Tips	Incompatibility of P256 Contract with Precompiled Contracts for Production Use	(Resolved)

OVERVIEW

Protocol overview

• **P256**

This Solidity library, P256, enables external contracts to verify P256 signatures, supporting a custom verifier (`DAIMO_VERIFIER`) and a more gas-efficient precompiled contract (`PRECOMPILED_VERIFIER`). It includes two functions for signature verification: `verifySignatureAllowMalleability` allows for signature malleability and uses either verifier based on the `usePrecompiled` flag. At the same time, `verifySignature` adds a malleability check to prevent signature replay by validating the s component against the P256 curve's half-order. Using a precompiled contract for verification is highlighted as a cost-effective alternative, optimizing gas consumption without compromising security and demonstrating a balance between efficiency and customizability in smart contract development.

• **P256Validator**

The `P256Validator` contract is designed to validate signatures using the P256 curve. It implements the `IKernelValidator` interface and includes functionality to enable and disable the validator for a kernel account. The contract stores P256 public keys for kernel accounts and provides functions to validate `UserOperation` data and signatures using these keys. The `validateUserOp` function validates a `UserOperation` data by verifying the signature against the corresponding public key, while the `validateSignature` function validates a signature directly.

• **WebAuthn**

The contract in the provided code is a library named WebAuthn, which serves as a helper for external contracts to verify WebAuthn signatures. It includes functions to check if a string contains a substring starting from a specified location, verify authentication flags in authenticator data, and verify a WebAuthn P256 signature by checking various components such as flags, client JSON type, challenge presence, and the validity of the signature over authData and client JSON with the provided public key coordinates. This contract assumes validity checks for the signature while delegating other responsibilities,

such as client extension outputs, attestation object verification, and signature counter validation, to external factors like the authenticator or relying party policies. The contract aims to provide a simplified verification process for WebAuthn signatures based on the specified criteria.

• **WebAuthnValidator**

This contract, named `WebAuthnValidator`, serves as an implementation of the `IKernelValidator` interface and is designed to validate WebAuthn signatures. It includes functionality to enable and disable the WebAuthn validator for a kernel account, validate `UserOperation` data, validate signatures, and verify signatures using stored public keys. The contract utilizes the WebAuthn library to verify signatures and provides methods for handling signature validation outcomes. The contract aims to enhance security by ensuring the authenticity and integrity of `UserOperation` data through signature validation.

• **Base64URL**

The contract in the provided code is a Solidity library named `Base64URL`. It includes an encode function that takes a bytes memory data parameter and returns a string memory. This function internally uses the Base64 library to encode the data and then modifies the output to conform to `Base64URL` standards.

Scope

```
src
├── P256.sol
├── P256Validator.sol
├── WebAuthn.sol
├── WebAuthnValidator.sol
└── utils
    └── Base64URL.sol
```

FINDINGS

1. Potential Gas Cost Manipulation via changing usePrecompiled flag

ID: ZeroDev-WebAuthn-01
Type: Gas Griefing
File: p256/src/WebAuthnValidator.sol

Severity: Low
Difficulty: Low

Issue

The decision to utilize the DAIMO verifier or a precompiled verifier for signature verification in `WebAuthnValidator` is determined by the usePrecompiled flag included within the `UserOperation`'s signature field. It has been identified that this flag can be externally altered due to its inclusion in a publicly accessible mempool. This vulnerability could allow malicious actors to change the `usePrecompiled` flag from the intended precompiled contract to the potentially more gas-intensive DAIMO onchain verifier, leading to higher gas costs. This risk does not necessarily provide a direct financial benefit to the attacker but could be exploited for sabotage, such as inflicting economic damage on competitors or the system itself. The likelihood of such issues occurring is very low as long as the calculation of the `verificationGasLimit` in `UserOperation` is accurate.

```
function _verifySignature(address sender, bytes32 hash, bytes calldata signature)
    private
    view
    returns (ValidationData)
{
    // decode the signature
    (
        bytes memory authenticatorData,
        string memory clientDataJSON,
        uint256 responseTypeLocation,
        uint256 r,
        uint256 s,
        bool usePrecompiled
    ) = abi.decode(signature, (bytes, string, uint256, uint256, uint256, bool));

    // get the public key from storage
    WebAuthnValidatorData memory pubKey = webAuthnValidatorStorage[sender];

    // verify the signature using the signature and the public key
```

```
    bool isValid = WebAuthn.verifySignature(
        abi.encodePacked(hash),
        authenticatorData,
        true,
        clientDataJSON,
        CHALLENGE_LOCATION,
        responseTypeLocation,
        r,
        s,
        pubKey.x,
        pubKey.y,
        usePrecompiled
    );

    // return the validation data
    if (isValid) {
        return ValidationData.wrap(0);
    }

    return SIG_VALIDATION_FAILED;
}
```

https://github.com/zerodevapp/kernel-plugins/blob/dfc53f96dc4513d3c410881adb722cc618878d47/p256/src/WebAuthnValidator.sol#L85-L124

Recommendation

To enhance security and ensure the integrity of the `usePrecompiled` flag, it is recommended to store the preference for using a precompiled contract within `webAuthnValidatorStorage` at the time of enabling the `WebAuthnValidator` through the enable function. This approach would make it significantly more challenging for unauthorized parties to alter the verification process.

Patch Comment

We have confirmed that the issue mentioned in this patch (https://github.com/zerodevapp/kernel-plugins/commit/fc1c0f15463016bc8e97bc28e4b835728487e53d) has been resolved.

2. Incompatibility of P256 Contract with Precompiled Contracts for Production Use

ID: ZeroDev-WebAuthn-02
Type: Incompatibility Contract
File: p256/src/P256Validator.sol

Severity: Tips
Difficulty: Tips

Issue

The `P256Validator` currently utilizes a P256 contract provided by DAIMO that does not support precompiled contracts for signature verification. This limitation is a significant concern for production deployment, where gas efficiency is critical. The absence of precompiled contract support could lead to higher operational costs and decreased system efficiency.

Recommendation

To prepare the `P256Validator` to improve gas efficiency, it is essential to update the implementation to include a P256 contract that supports the use of precompiled contracts for signature verification.

Patch Comment

We have confirmed that the issue mentioned in this patch (https://github.com/zerodevapp/kernel-plugins/commit/fc1c0f15463016bc8e97bc28e4b835728487e53d) has been resolved.

DISCLAIMER

This report does not guarantee investment advice, the suitability of the business models, and codes that are secure without bugs. This report shall only be used to discuss known technical issues. Other than the issues described in this report, undiscovered issues may exist such as defects on the main network. In order to write secure codes, correction of discovered problems and sufficient testing thereof are required.

Appendix. A

Severity Level

CRITICAL	Must be addressed as a vulnerability that has the potential to seize or freeze substantial sums of money.
HIGH	Has to be fixed since it has the potential to deny users compensation or momentarily freeze assets.
MEDIUM	Vulnerabilities that could halt services, such as DoS and Out-of-Gas, need to be addressed.
LOW	Issues that do not comply with standards or return incorrect values
TIPS	Tips that makes the code more usable or efficient when modified

Difficulty Level

	Low	Medium	High
Privilege	anyone	Miner/Block Proposer	Admin/Owner
Capital needed	Small or none	Gas fee or volatile as price change	More than exploited amount
Probability	100%	Depend on environment	Hard as mining difficulty

Vulnerability Category

Arithmetic	• Integer under/overflow vulnerability • floating point and rounding accuracy
Access & Privilege Control	• Manager functions for emergency handle • Crucial function and data access • Count of calling important task, contract state change, intentional task delay
Denial of Service	• Unexpected revert handling • Gas limit excess due to unpredictable implementation
Miner Manipulation	• Dependency on the block number or timestamp. • Frontrunning
Reentrancy	•Proper use of Check-Effect-Interact pattern. •Prevention of state change after external call • Error handling and logging.
Low-level Call	• Code injection using delegatecall • Inappropriate use of assembly code
Off-standard	• Deviate from standards that can be an obstacle of interoperability.
Input Validation	• Lack of validation on inputs.
Logic Error/Bug	• Unintended execution leads to error.
Documentation	•Coherency between the documented spec and implementation
Visibility	• Variable and function visibility setting
Incorrect Interface	• Contract interface is properly implemented on code.

End of Document