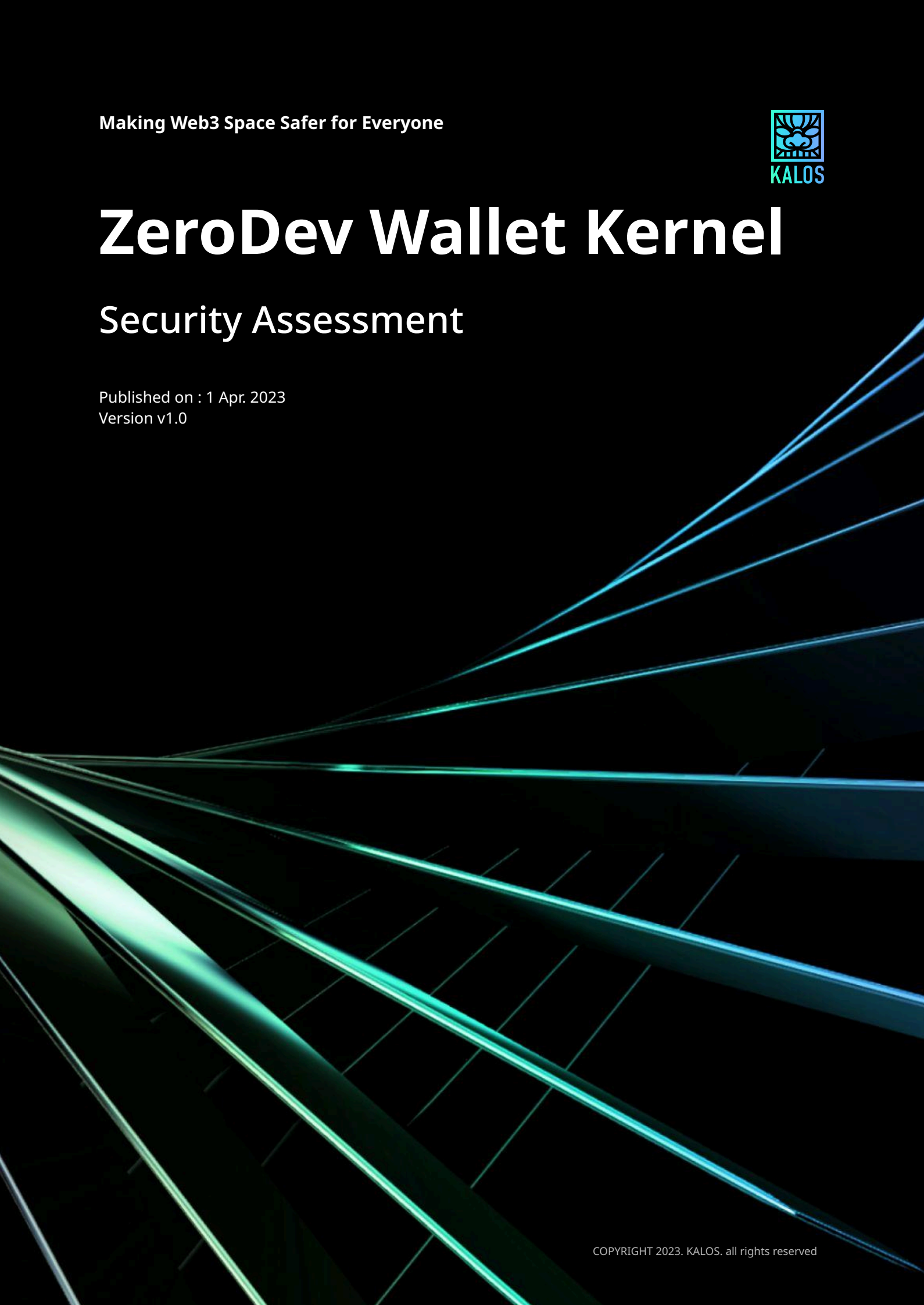
Making Web3 Space Safer for Everyone

KALOS

ZeroDev Wallet Kernel

Security Assessment

Published on : 1 Apr. 2023
Version v1.0



Security Report Published by KALOS

v1.0 1 Apr. 2023

Auditor : Paul Kim, Jade Han

Found issues

Severity of Issues	Findings	Resolved	Acknowledged	Comment
Critical	-	-	-	-
High	-	-	-	-
Medium	-	-	-	-
Low	-	-	-	-
Tips	3	3	-	-

TABLE OF CONTENTS

ABOUT US

Making Web3 Space Safer for Everyone

KALOS is a flagship service of HAECHI LABS, the leader of the global blockchain industry. We bring together the best Web2 and Web3 experts. Security Researchers with expertise in cryptography, leaders of the global best hacker team, and blockchain/smart contract experts are responsible for securing your Web3 service.

Having secured $60B crypto assets on over 400 main-nets, Defi protocols, NFT services, P2E, and Bridges, KALOS is the only blockchain technology company selected for the Samsung Electronics Startup Incubation Program in recognition of our expertise. We have also received technology grants from the Ethereum Foundation and Ethereum Community Fund.

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

Codebase Submitted for the Audit

The codes used in this Audit can be found on GitHub (https://github.com/ZeroDevapp/ZeroDev-wallet-kernel).

The commit of the code used for this Audit is “ae356c5478fc7c500bf270d864ef5eee70f292e6”,

Audit Timeline

Date	Event
2023/03/28	Audit Initiation (ZeroDev wallet kernel)
2023/04/03	Delivery of v1.0 report.

Findings

KALOS found - 0 Critical, 0 High, 0 medium, 0 Low and 3 tips severity issues.

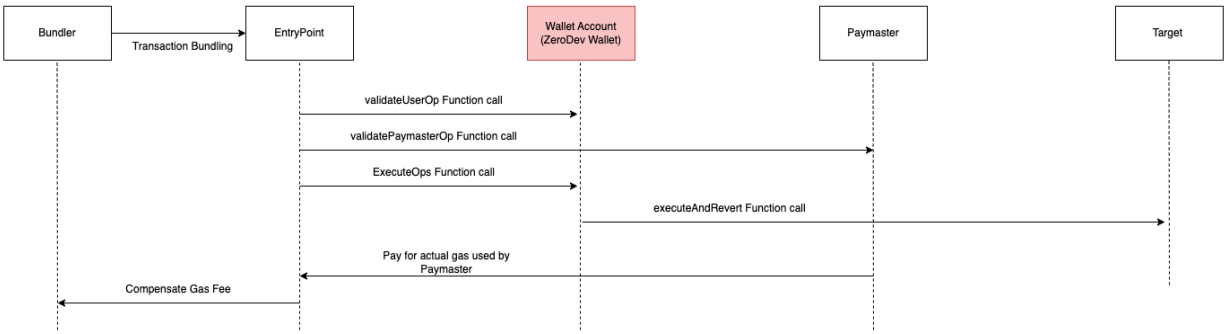
Severity	Issue	Status
Tips	Replay attack can be caused on MinimalAccount	(Fixed - v1.0)
Tips	Replay attack can be caused on validateUserOp of Kernel	(Fixed - v1.0)
Tips	checkUserOpOffset has the wrong condition checking logic	(Fixed - v1.0)

Remarks

The solidity code within the plugins folder is not in audit scope.

OVERVIEW

Protocol overview



The image above is a high-level overview of the ERC-4337 Contract and shows a rough structure of how the Wallet Account developed by ZeroDev interacts with the other elements of ERC-4337.

• **Compatibility.sol**

The contract is an abstract contract and it defines several functions that are designed to make the contract compatible with other contracts that follow certain standards.

The first function is a fallback function that allows the contract to receive ether.

The "onERC721Received" function is a callback function that is called when the contract receives an ERC721 token. It returns the selector for the function as a bytes4 value.

The "onERC1155Received" function is a similar callback function that is called when the contract receives an ERC1155 token. It also returns the selector for the function as a bytes4 value.

The "onERC1155BatchReceived" function is another callback function that is called when the contract receives multiple ERC1155 tokens. It returns the selector for the function as a bytes4 value.

Finally, the "isValidSignature" function is a function that can be used to check the validity of a signature for a given hash. It returns the selector for the function as a bytes4 value.

Overall, this contract is designed to ensure compatibility with the ERC721 and ERC1155 token standards, and provides a signature verification function for added security.

• **KernelStorage.sol**

This Solidity contract defines a kernel store consisting of an owner and a nonce.

The contract contains functions to get the owner and nonce of the kernel storage, and functions to upgrade the implementation of the contract.

It is inherited by the MinimalAccount Contract and the Kernel Contract.

• **AccountFactory.sol**

This contract is an implementation of an account factory, which allows anyone to create new accounts that are replicas of a predefined account template. The account template is a contract called `MinimalAccount` that is passed to the factory's constructor during deployment.

The `createAccount` function creates a new account by computing a unique address using the `Create2.computeAddress` function with a combination of the `_owner` and `_index` parameters. The generated address is checked to see if any contract code is already deployed. If the code exists, it returns the existing contract address. Otherwise, it clones the account template and deploy a new contract using the `EIP1967Proxy` contract and update the owner by calling the initialize function.

The `getAccountAddress` function uses `Create2.computeAddress` to compute the address of the pre-deployed Account Contract.

Overall, this contract allows anyone to create new accounts efficiently and securely, saving used gas compared to deploying a new contract every time.

• **EIP1967Proxy.sol**

The `EIP1967Proxy` contract is an implementation of the EIP-1967 standard for upgradable smart contracts. It allows the user to delegate all function calls to another contract, called the implementation contract, and at the same time upgrade the implementation contract at any time without disturbing the state of the contract.

The `EIP1967Proxy` contract has a single storage slot `_IMPLEMENTATION_SLOT` that stores the address of the current implementation contract. When a function is called from

the `EIP1967Proxy` contract, it uses the `delegatecall()` function to forward the call to the current implementation contract.

This way, the called function executes the logic of the implementation contract, but in the context of the proxy contract, and all state changes are made in the proxy contract.

The `EIP1967Proxy` contract is designed to be deployed once and then upgrade the implementation contract as needed. To upgrade an implementation contract, the user deploys a new implementation contract and then updates the `_IMPLEMENTATION_SLOT` storage slot to point to the address of the new implementation contract.

Overall, the `EIP1967Proxy` contract provides a way to create upgradable smart contracts with minimal disruption to the contract's state and functionality.

• **MinimalAccount.sol**

The `MinimalAccount` contract is a minimalistic implementation of the `IAccount` interface which provides basic account management functionalities. It implements a subset of the functions required for an account in a modular and reusable manner that can be incorporated into other contracts.

This contract allows for the execution of function calls to external contracts and it also provides a function to verify signatures using the EIP1271 standard. The `executeAndRevert` function is used to execute function calls to external contracts while `isValidSignature` is used to verify signatures.

The `initialize` function initializes the contract owner and can only be called once. The `validateUserOp` function validates a user operation and checks if the signature is valid. It also allows for the forwarding of amounts owed to the entryPoint Contract.

Overall, the `MinimalAccount` contract provides a simple and modular way to implement account management functionalities that can be easily integrated into other contracts.

• **Exec.sol**

The contract is a Solidity library that provides utility functions for making different kinds of contract calls. It contains three functions: `call`, `staticcall`, and `delegateCall`.

The `call` function is used to make a regular call to another contract, allowing it to execute any function in that contract and return any data. The function takes three parameters: the address of the contract to call, the amount of ether to send in the call (if any), and the data to send.

The `staticcall` function is similar to `call`, with the key difference being that it does not allow the called contract to modify the state of the current contract. It is used for read-only calls, such as getting the current value of a contract's variable.

The `delegateCall` function is used to make a delegate call to another contract, allowing it to execute any function in that contract on behalf of the current contract. This means that any state changes made by the called contract will be made to the current contract's storage. The function takes two parameters: the address of the contract to call, and the data to send.

All three functions are implemented using inline assembly, and they return a boolean indicating whether the call was successful, as well as any data returned by the called contract. The contract also defines an `Operation` enum with two values: `Call` and `DelegateCall`.

• **ExtendedUserOpLib.sol**

The contract in the code is a library named `ExtendedUserOpLib` that provides a function named `checkUserOpOffset`.

This function is used for handling a vulnerability in the `UserOperation` interface of the `account-abstraction` library, which is included as an imported dependency.

The vulnerability in question is related to the fact that the `UserOperation` interface uses offsets to locate data within the transaction payload, which can be manipulated to cause unintended behavior. The `checkUserOpOffset` function provides a fix for this vulnerability by checking that the offsets used by the various data fields in the `UserOperation` struct are in the correct order.

Overall, the contract serves as a security enhancement to the `account-abstraction` library by providing a fix for a known vulnerability.

• **Kernel.sol**

The `Kernel` contract is a minimalist wallet core that supports only one owner and multiple plugins. It allows users to query plugins for data and execute function calls to external contracts. It also validates user operations and signatures, and provides a way for plugins to verify user operations.

The contract includes the following functions:

- `initialize`: Initializes the wallet kernel and can only be called once. Takes the owner's address as a parameter.

- `queryPlugin`: Queries a plugin for data and returns an error message if the query is successful. This function is used to simulate changing on-chain storage.

- `executeAndRevert`: Executes a function call to an external contract and reverts if it is unsuccessful. Takes the target contract address, value, data, and operation type as parameters.

- `validateUserOp`: Validates a user action and returns the validation data. Takes as parameters the user action, the user action hash, and the amount owed to the entryPoint Contract.

- `isValidSignature`: Validates a signature using EIP1271 and returns a `bytes4` value indicating the validity of the signature.

The contract inherits from the `IAccount`, `EIP712`, `Compatibility`, and `KernelStorage` contracts, which provide functionality for account management, EIP-712 type and data definitions, plugin and user operation compatibility checks, and kernel storage management, respectively.

Overall, the `Kernel` contract acts as a core wallet module that provides basic functionality and enables plugins to extend its capabilities.

Scope

```
├── Kernel.sol
├── abstract
│   ├── Compatibility.sol
│   └── KernelStorage.sol
├── factory
│   ├── AccountFactory.sol
│   ├── EIP1967Proxy.sol
│   └── MinimalAccount.sol
└── utils
    ├── Exec.sol
    └── ExtendedUserOpLib.sol
```

Access Controls

Access control in contracts is achieved using modifiers and inline require statements.
The access control of Stablecomp refers to the following variables.

- ❖ entryPoint
- ❖ owner

entryPoint : A relay contract that executes the functions of the Wallet Contract.

- KernelStorage.sol#upgradeTo(address)
- MinimalAccount.sol#validateUserOp(UserOperation calldata, bytes32, uint256)
- MinimalAccount.sol#executeAndRevert(address, uint256, bytes calldata, Operation)
- Kernel.sol#validateUserOp(UserOperation calldata, bytes32, uint256)
- Kernel.sol#executeAndRevert(address, uint256, bytes calldata, Operation)

owner : The wallet address of the beneficial owner of the Wallet Contract.

- KernelStorage.sol#upgradeTo(address)
- MinimalAccount.sol#validateUserOp(UserOperation calldata, bytes32, uint256)
- MinimalAccount.sol#executeAndRevert(address, uint256, bytes calldata, Operation)
- Kernel.sol#validateUserOp(UserOperation calldata, bytes32, uint256)
- Kernel.sol#executeAndRevert(address, uint256, bytes calldata, Operation)

FINDINGS

Signature Replay Attack can be done on MinimalAccount

ID: ZeroDev-wallet-01
Severity: Tips
Type: Logic Error
Difficulty: Low
File: src/factory/MinimalAccount.sol

Issue

For the MinimalAccount, the nonce is increased only when `userOp.initCode` exists. This means, the nonce will not be increased when the callData exists but not initCode. Therefore, the attacker can replay the UserOperation.

```
function validateUserOp(UserOperation calldata userOp, bytes32 userOpHash,
uint256 missingFunds)
        external
        returns (uint256)
    {
        require(ExtendedUserOpLib.checkUserOpOffset(userOp), "Invalid userOp");
        require(msg.sender == address(entryPoint), "account: not from
entrypoint");
        bytes32 hash = ECDSA.toEthSignedMessageHash(userOpHash);
        address recovered = ECDSA.recover(hash, userOp.signature);
        WalletKernelStorage storage ws = getKernelStorage();
        if (ws.owner != recovered) {
            return SIG_VALIDATION_FAILED;
        }
        if (userOp.initCode.length > 0) {
            if (ws.nonce++ != userOp.nonce) {
                revert InvalidNonce();
            }
        }

        if (missingFunds > 0) {
            (bool success,) = msg.sender.call{value: missingFunds}("");
            (success);
        }
        return 0;
    }
```

[https://github.com/ZeroDevapp/ZeroDev-wallet-kernel/blob/main/src/factory/MinimalAccount.sol#L23-L45]

Recommendation

We recommend to increase the nonce not only when there is an initCode but also when there is a callData.

Fix Comment

EntryPoint has a nonce that prevents replay attacks, so project team removed the code related to the nonce in the Wallet implementation.
(https://github.com/eth-infinitism/account-abstraction/pull/247/files#diff-68a757b8507b386b2a26bf4381908da3b868ffc4cb9d1b95e3162bb8fe38a869)
In the end, this code is safe from replay attacks.

Signature Replay Attack can be done on validateUserOp of Kernel

ID: ZeroDev-wallet-02
Type: Logic Error
File: src/Kernel.sol

Severity: Tips
Difficulty: Low

Issue

The Kernel has two cases following the signature. When the user uses a plugin, the signature must be larger than length 97. So, the following code will be executed.

```
        } else if (userOp.signature.length > 97) {
            // userOp.signature = address(plugin) + validUntil + validAfter +
pluginData + pluginSignature
            address plugin = address(bytes20(userOp.signature[0:20]));
            uint48 validUntil = uint48(bytes6(userOp.signature[20:26]));
            uint48 validAfter = uint48(bytes6(userOp.signature[26:32]));
            bytes memory signature = userOp.signature[32:97];
            (bytes memory data,) = abi.decode(userOp.signature[97:], (bytes,
bytes));

            bytes32 digest = _hashTypedDataV4(
                keccak256(
                    abi.encode(
                        keccak256(
                            "ValidateUserOpPlugin(address sender,uint48
validUntil,uint48 validAfter,address plugin,bytes data)"
                        ), // we are going to trust plugin for verification
                        userOp.sender,
                        validUntil,
                        validAfter,
                        plugin,
                        keccak256(data)
                    )
                )
            );

            address signer = ECDSA.recover(digest, signature);
            if (getKernelStorage().owner != signer) {
                return SIG_VALIDATION_FAILED;
            }
            bytes memory ret = _delegateToPlugin(plugin, userOp, userOpHash,
missingAccountFunds);
            bool res = abi.decode(ret, (bool));
```

```
            if (res) {
                return SIG_VALIDATION_FAILED;
            }
            validationData = _packValidationData(!res, validUntil, validAfter);
```

[https://github.com/ZeroDevapp/ZeroDev-wallet-kernel/blob/main/src/Kernel.sol#L94-L124]

We checked the nonce and found no increase in nonce, but the ZeroDev team was already aware of the issue and informed us that they would advise plugin developers to include signature replay protection code within their plugins, so we mapped the issue to a tip.

Recommendation

We recommend that the code increases the nonce when per UserOperation is executed.

Fix Comment

The ZeroDev team will advise plugin developers to include signature replay protection code within their plugins.

Wrong condition checking logic in checkUserOpOffset

ID: ZeroDev-wallet-03
Type: Logic Error
File: src/utils/ExtendedUserOpLib.sol

Severity: Tips
Difficulty: Low

Issue

The function checkUserOpOffset of the ExtendedUserOpLib library is there to check the proper offset of given UserOperation. The success return value is set as 1 even if the userOp passes the first condition only. So, the second and third condition will be ignored.

```
    function checkUserOpOffset(UserOperation calldata userOp) internal pure
returns (bool success) {
        bytes calldata sig = userOp.signature;
        bytes calldata cd = userOp.callData;
        bytes calldata initCode = userOp.initCode;
        bytes calldata paymasterAndData = userOp.paymasterAndData;
        assembly {
            if eq(add(initCode.offset, mul(div(add(initCode.length, 63), 32),
0x20)), cd.offset) { success := 1 }
            if and(eq(add(cd.offset, mul(div(add(cd.length, 63), 32), 0x20)),
paymasterAndData.offset), success) {
                success := 1
            }
            if and(
                eq(add(paymasterAndData.offset,
mul(div(add(paymasterAndData.length, 63), 32), 0x20)), sig.offset),
                success
            ) { success := 1 }
        }
    }
```

[https://github.com/ZeroDevapp/ZeroDev-wallet-kernel/blob/main/src/utils/ExtendedUserOpLib.sol#L9-L24]

Recommendation

We found the ExtendedUserOpLib.checkUserOpOffset is unnecessary for the Kernel contract. This function is implemented to prevent the vulnerability which is stated on https://github.com/eth-infinitism/account-abstraction/issues/237. The vulnerability of this link can be caused by limited code structure only. We analyzed that the ZeroDev team's code is not affected by that vulnerability. So, we recommend removing this function.

Fix Comment

The ZeroDev team decided to remove this function.

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