Making Web3 Space Safer for Everyone

KALOS

ZeroDev Wallet Kernel V2.1-Lite

Security Assessment

Published on : 20 Sep. 2023
Version v2.0

Security Report Published by KALOS

v2.0 20 Sep. 2023

Auditor : Jade Han

hojung han

Found issues

Severity of Issues	Findings	Resolved	Acknowledged	Comment
Critical	1	1	-	-
High	-	-	-	-
Medium	-	-	-	-
Low	-	-	-	-
Tips	-	-	-	-

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
- Compatibility Testing with Bundler

Codebase Submitted for the Audit

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

The commit hash of the code used for this Audit is “f00ee0fb519a3d85a5814c0b2fecfe476db36ed0”.

The commit hash of the patched according to our recommendations is “749338ac5ee6c0016cdf9f0c0c5c8a57688d4845”

Audit Timeline

Date	Event
2023/07/31	Audit Initiation (ZeroDev wallet kernel V2.1)
2023/08/09	Delivery of v1.0 report.

2023/09/18	Audit Initiation (ZeroDev wallet kernel V2.1 Lite)
2023/09/20	Delivery of v2.0 report.

Findings

KALOS found - 1 Critical, 0 High, 0 medium, 0 Low and 0 tips severity issues.

Severity	Issue	Status
Critical	Incomplete Initialization of the Owner Address in KernelLiteECDSA Implementation Contract	(Resolved)

OVERVIEW

Kernel V2.1-Lite

The Kernel V2.1-Lite continues to inherit most features of Kernel V2.1, including implementing the Validator and Execution modules.

The Validator module still maintains its role in validating the provided signature, enabling additional requirement checks, such as the ability for wallet owners to limit supported function selectors. The Execution module also retains its functionality, providing developers the flexibility and security to add functions to the Kernel and pair them with validation plugins.

The KernelV2.1-Lite differs from KernelV2.1 in that it cannot change the defaultValidator, but all other functions result identically to KernelV2.1.

Functional Description

Validation Mode

The Zerodev Kernel V2.1-Lite has three validation modes. Each of these modes provides different levels of control and permissions within the Kernel, allowing developers to manage execution and validation in a way that suits their specific use-case.

- Sudo Mode: This mode is used to execute any function currently registered in the Kernel. It can only use the defaultValidator, which has the most permission on the Kernel. In Kernel v2.1-Lite, unlike Kernel v2.1, the Wallet Owner cannot change the defaultValidator and continues to use the defaultValidator set initially. (0x00000000)
- Plugin Mode: This mode is automatically enabled with the function selector of the userOp.callData, which should be set during the plugin registration process. Just like Sudo mode, all it needs is the signature that will be used on the validator. (0x00000001)
- Enable Mode: This mode is for setting the plugin during the validation phase to minimize the flow of using the plugin. Enable mode requires the signature to be packed in a certain way. This mode will set the execution data and enable the validator with enableData. The enableSignature should be the signature used by

defaultValidator following EIP712 using typeHash of ValidatorApproved(bytes4 sig,uint256 validatorData,address executor,bytes enableData).(0x00000002)

Validator

- ECDSAValidator: Designed to authenticate transactions. It establishes an owner during initialization and allows for owner removal. It validates operations and signatures by comparing the owner to the address recovered from a given hash and signature. If the addresses match, validation is successful; otherwise, it fails.
- ERC165SessionKeyValidator: Combines the concept of session key and the ERC165 standard to manage and validate the operations, with the additional ability to perform interface checks based on the ERC165 standard. The session key is a trusted third party with limited permissions, often used for specific actions like transferring ERC721 tokens. ERC165 is a standard used to verify if a particular interface is implemented by a given address.
- KillSwitchValidator: Similarly to an ECDSA Validator, primarily verifying that operations are signed by the owner. It also introduces a recovery mechanism, where a designated guardian can pause operations and assign a new owner if recovery is needed. After the set pause period, the new owner can use the system freely. This enhances the security by adding an extra layer of protection and a safety net for the owner.
- MultiECDSAValidator: A sophisticated contract designed to enhance operational security through the implementation of multi-ownership. This contract enables the addition and removal of owners and ensures only transactions from valid owners are processed. Each owner is tied to a specific kernel and can be dynamically managed through the 'disable' and 'enable' functions. The contract leverages the ECDSA cryptographic algorithm for signature verification, ensuring transaction integrity. The validation of user operations is carried out by the validateUserOp function, and validateSignature function checks the signature against the owner list. The validCaller function ensures only legitimate owners can call operations, increasing security against potential unauthorized access.
- SessionKeyValidator.sol: A complex contract designed to regulate operations via session keys with pre-set validity periods, a Merkle root, and bespoke validation parameters. The 'enable' function initiates a session key, setting its valid period and a Merkle root. Contrarily, the 'disable' function removes the session key,

terminating its operations. The contract's 'validateUserOp' function performs meticulous checks on each operation, ensuring that each session key is valid and active. Furthermore, it examines the associated parameters, such as target address, value, signature, etc., for the operation, verifying their compliance with the permissions stipulated in the Merkle root through a Merkle proof.

Factory

Within the Scope, two contracts are utilized to create the KernelLite Account: AdminLessERC1967Factory, KernelFactory.

The KernelFactory Contract inherits from the AdminLessERC1967Factory Contract created by solady. The Kernel Factory includes the functions of the AdminLessERC1967Factory Contract and is written to ensure compatibility with ERC-4337.

Scope

```
src
├── Kernel.sol
├── KernelLite.sol
├── abstract
│   ├── Compatibility.sol
│   └── KernelStorage.sol
├── common
│   ├── Constants.sol
│   ├── Enum.sol
│   ├── Structs.sol
│   └── Types.sol
├── executor
│   ├── KillSwitchAction.sol
│   └── TokenActions.sol
├── factory
│   ├── AdminLessERC1967Factory.sol
│   └── KernelFactory.sol
├── interfaces
│   ├── IAddressBook.sol
│   ├── IKernel.sol
│   └── IValidator.sol
├── lite
│   └── KernelLiteECDSA.sol
├── test
│   ├── TestCounter.sol
│   ├── TestERC20.sol
│   ├── TestERC721.sol
│   ├── TestExecutor.sol
│   ├── TestKernel.sol
│   └── TestValidator.sol
├── utils
│   └── KernelHelper.sol
└── validator
    ├── ECDSAValidator.sol
    ├── ERC165SessionKeyValidator.sol
    ├── KillSwitchValidator.sol
    ├── MultiECDSAValidator.sol
    ├── SessionKeyOwnedValidator.sol
    └── SessionKeyValidator.sol
```

*** We have verified whether the codes within the Scope are sufficiently compatible with the 4337 Specification using the Bundler (https://github.com/pimlicolabs/eth-infinitism-bundler).**

Access Controls

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

- ❖ entryPoint
- ❖ Validated Signer (Owner, Session)

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

- KernelLite.sol#fallback
- KernelLite.sol#execute
- KernelLite.sol#executeBatch
- KernelLite.sol#validateUserOp
- KernelStorage.sol#upgradeTo
- KernelStorage.sol#setExecution
- KernelStorage.sol#setDefaultValidator
- KernelStorage.sol#disableMode

Validated Signer (Owner, Session) : The address of the allowed signer of UserOperation. This account can make a call to a predefined executor or target address. Under the premise that the session has received permission from the Owner, it can execute the following functions within certain limits.

- Kernel.sol#fallback
- Kernel.sol#execute
- Kernel.sol#executeBatch

FINDINGS

1. Incomplete Initialization of the Owner Address in KernelLiteECDSA Implementation Contract

ID: ZeroDev-2-1
Severity: Critical
Type: Improper Initialization
Difficulty: Low
File: src/lite/KernelLiteECDSA.sol

Issue

The `KernelStorage` contract uses its constructor to set the `getKernelStorage().defaultValidator` to `address(1)` mitigate to prevent execution of the initialize function in the `KernelLiteECDSA` implementation contract.

```
constructor(IEntryPoint _entryPoint) {
    entryPoint = _entryPoint;
    getKernelStorage().defaultValidator = IKernelValidator(address(1));
}
```

https://github.com/zerodevapp/kernel/blob/f00ee0fb519a3d85a5814c0b2fecfe476db36ed0/src/abstract/KernelStorage.sol#L36-L39

However, the `KernelLiteECDSA` contract contains `_setInitialData` function, which checks if the owner's address in `getKernelLiteECDSAStorage()` is `address(0)` before setting the owner's address. The storage slots being dealt with in `getKernelStorage()` and `getKernelLiteECDSAStorage()` are different.

```
function _setInitialData(IKernelValidator, bytes calldata _data) internal override {
    require(getKernelLiteECDSAStorage().owner == address(0), "KernelLiteECDSA: already
initialized");
    address owner = address(bytes20(_data[0:20]));
    getKernelLiteECDSAStorage().owner = owner;
}
```

https://github.com/zerodevapp/kernel/blob/f00ee0fb519a3d85a5814c0b2fecfe476db36ed0/src/lite/KernelLiteECDSA.sol#L21-L25

Consequently, the mitigation in the former doesn't effectively prevent potential vulnerabilities in `KernelLiteECDSA` implementation. Suppose an attacker manipulates the owner's address in the `KernelLiteECDSA` implementation and executes `selfdestruct` using `delegateCall`.

In that case, it cause issues in the proxy contract that references this implementation.

Recommendation

It's imperative to initialize the `getKernelLiteECDSAStorage().owner` within the constructor of the `KernelLiteECDSA` contract to a value other than `address(0)`. This change ensures proper mitigation against this vulnerability associated with the improper initialization of contract owner addresses.

Patch Comment

We have verified that the code has been patched according to our recommendations.
(https://github.com/zerodevapp/kernel/commit/3112be29c07360c0beff5548ec1d2a002a23032b)

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