Making Web3 Space Safer for Everyone

KALOS

ZeroDev Wallet Kernel V2.1

Security Assessment

Published on : 09 Aug. 2023
Version v2.1

Security Report Published by KALOS

v2.1 09 Aug. 2023

Auditor : Jade Han

hojung han

Found issues

Severity of Issues	Findings	Resolved	Acknowledged	Comment
Critical	1	1	-	-
High	2	1	1	-
Medium	4	4	-	-
Low	-	-	-	-
Tips	1	1	-	-

TABLE OF CONTENTS

TABLE OF CONTENTS
ABOUT US
Executive Summary
OVERVIEW
- Kernel V2.1
- Functional Description
- Scope
- Access Controls

FINDINGS
- 1. Violation of Storage Access Rule in ECDSAKernelFactory
  - Issue
  - Recommendation
  - Patch Comment
- 2. Bypass Permission Check in ValidateUserOp of the SessionKeyValidator Contract
  - Issue
  - Recommendation
  - Patch Comment
- 3. Wrong Parameter for _packValidationData upon Successful Merkle Proof in validateUserOp of SessionKeyValidator
  - Issue
  - Recommendation
  - Patch Comment
- 4. Wrong dataOffset Calculation in ValidateUserOp within SessionKeyValidator
  - Issue
  - Recommendation
  - Patch Comment
- 5. Discrepancies in Parsing validUntil and validAfter
  - Issue
  - Recommendation
  - Patch Comment
- 6. Replay Attack in Setting DefaultValidator in KillSwitchValidator
  - Issue
  - Recommendation

Patch Comment
7. Manipulation of defaultValidator and Execution Information in Session Leakage Scenario
Issue
Recommendation
Patch Comment
8. DelegateCall Risk
Issue
Recommendation
Patch Comment
DISCLAIMER
Appendix. A
Severity Level
Difficulty Level
Vulnerability Category

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 “199ae7d838f21b37f069267c86e8eeb6f0175a69”,

Audit Timeline

Date	Event
2023/07/31	Audit Initiation (ZeroDev wallet kernel V2.1)
2023/08/09	Delivery of v2.1 report.

Findings

KALOS found - 1 Critical, 2 High, 4 medium, 0 Low and 1 tips severity issues.

Severity	Issue	Status
High	Violation of Storage Access Rule in ECDSAKernelFactory	(Resolved - v2.1)
Medium	Bypass Permission Check in ValidateUserOp of the SessionKeyValidator Contract	(Resolved - v2.1)
Medium	Wrong Parameter for _packValidationData upon Successful Merkle Proof in validateUserOp of SessionKeyValidator	(Resolved - v2.1)
Medium	Wrong dataOffset Calculation in ValidateUserOp within SessionKeyValidator	(Resolved - v2.1)
Medium	Discrepancies in Parsing validUntil and validAfter	(Resolved - v2.1)
Critical	Replay Attack in Setting DefaultValidator in KillSwitchValidator	(Resolved - v2.1)
High	Manipulation of defaultValidator and Execution Information in Session Leakage Scenario	(Acknowledged - v2.1)
Tips	DelegationCall Risk	(Resolved - v2.1)

OVERVIEW

Kernel V2.1

Kernel V2.1 continues to inherit most features of Kernel V2, 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.

Additions to Kernel V2.1 include the feature of directly utilizing the wallet without having to go through the Bundler. This development offers increased flexibility and user control. Furthermore, Kernel V2.1 has made modifications for gas optimization and better compliance with the 4337 Specification.

Functional Description

Validation Mode

The Zerodev Kernel V2.1 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. (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, four contracts are utilized to create the Kernel Account: AdminLessERC1967Factory, ECDSAKernelFactory, MultiECDSAKernelFactory, and KernelFactory.

There are two distinct pathways through which a Kernel Account is created via the EntryPoint:

1. ECDSAKernelFactory → KernelFactory → AdminLessERC1967Factory
2. MultiECDSAKernelFactory → KernelFactory → AdminLessERC1967Factory

In the first case, the newly created Kernel Account has its defaultValidator set to ECDSAValidator. In the second case, the defaultValidator of the new Kernel Account is set to MultiECDSAValidator.

Scope

```
src
├── Kernel.sol
├── abstract
│   ├── Compatibility.sol
│   └── KernelStorage.sol
├── executor
│   ├── KillSwitchAction.sol
│   └── TokenActions.sol
├── factory
│   ├── AdminLessERC1967Factory.sol
│   ├── ECDSAKernelFactory.sol
│   ├── KernelFactory.sol
│   └── MultiECDSAKernelFactory.sol
├── interfaces
│   ├── IAddressBook.sol
│   ├── IKernel.sol
│   └── IValidator.sol
├── test
│   ├── TestCounter.sol
│   ├── TestERC721.sol
│   ├── TestExecutor.sol
│   └── TestValidator.sol
├── utils
│   ├── Exec.sol
│   └── KernelHelper.sol
└── validator
    ├── ECDSAValidator.sol
    ├── ERC165SessionKeyValidator.sol
    ├── KillSwitchValidator.sol
    ├── MultiECDSAValidator.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.

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

FINDINGS

1. Violation of Storage Access Rule in ECDSAKernelFactory

ID: ZeroDevV2-1-01
Type: Logic Error
File: src/factory/ECDSAKernelFactory.sol

Severity: High
Difficulty: Low

Issue

According to the ERC 4337 Specification, the ECDSAKernelFactory is required to stake a certain amount in the EntryPoint due to the Storage Access rules.

If this is not done, the Bundler will not execute transactions due to rule violations. After setting up and testing the Bundler, the following error message occurred.

Recommendation

We recommend modifying the code to not violate the ERC4337 Specification by staking a certain amount in the EntryPoint through the addStake function.

Patch Comment

We have observed that the Factory Contract responsible for deploying KernelAccount has incorporated methods related to staking. (https://github.com/zerodevapp/kernel/pull/23)

2. Bypass Permission Check in ValidateUserOp of the SessionKeyValidator Contract

ID: ZeroDevV2-1-02
Severity: Medium
Type: Logic Error
Difficulty: Low
File: src/validator/SessionKeyValidator.sol

Issue

The ValidateUserOp of the SessionKeyValidator Contract appears to use a struct named Permission to limit user parameters within a certain range when calling a function of a specific contract without having to control each parameter.

However, to use the Permission struct as intended, the leaf should not be valued as keccak256(abi.encodePacked(target, value, data)) when performing a Merkle Proof. This statement does not check the parameters within the specified range but makes the target, value, and data into a Merkle Tree Leaf as used in v2.

```
function validateUserOp(UserOperation calldata userOp, bytes32 userOpHash, uint256 missingFunds)
   external
   payable
   returns (uint256)
{
   ...

   (Permission memory permission, bytes32[] memory merkleProof) =
       abi.decode(userOp.signature[65:], (Permission, bytes32[]));
   address target = address(bytes20(userOp.callData[16:36]));
   require(target == permission.target, "SessionKeyValidator: target mismatch");
   uint256 value = uint256(bytes32(userOp.callData[36:68]));
   require(value <= permission.valueLimit, "SessionKeyValidator: value limit exceeded");
   uint256 dataOffset = uint256(bytes32(userOp.callData[68:100]));
   uint256 dataLength = uint256(bytes32(userOp.callData[dataOffset:dataOffset + 32]));
   bytes calldata data = userOp.callData[dataOffset + 32:dataOffset + 32 + dataLength];
   bytes4 sig = bytes4(data[0:4]);
   require(sig == permission.sig, "SessionKeyValidator: sig mismatch");
   for (uint256 i = 0; i < permission.rules.length; i++) {
       ParamRule memory rule = permission.rules[i];
       bytes32 param = bytes32(data[4 + rule.index * 32:4 + rule.index * 32 + 32]);
       if (rule.condition == ParamCondition.EQUAL) {
           require(param == rule.param, "SessionKeyValidator: param mismatch");
       } elsev if (rule.condition == ParamCondition.GREATER_THAN) {
           require(param > rule.param, "SessionKeyValidator: param mismatch");
       } else if (rule.condition == ParamCondition.LESS_THAN) {
           require(param < rule.param, "SessionKeyValidator: param mismatch");
       } else if (rule.condition == ParamCondition.GREATER_THAN_OR_EQUAL) {
           require(param >= rule.param, "SessionKeyValidator: param mismatch");
       } else if (rule.condition == ParamCondition.LESS_THAN_OR_EQUAL) {
           require(param <= rule.param, "SessionKeyValidator: param mismatch");
       } else if (rule.condition == ParamCondition.NOT_EQUAL) {
```

```
            require(param != rule.param, "SessionKeyValidator: param mismatch");
        }
    }
    bytes32 leaf = keccak256(abi.encodePacked(target, value, data));
    bool result = MerkleProofLib.verify(merkleProof, session.merkleRoot, leaf);
    return _packValidationData(result, session.validUntil, session.validAfter);
}
```

https://github.com/zerodevapp/kernel/tree/199ae7d838f21b37f069267c86e8eeb6f0175a69/src/validator/SessionKeyValidator.sol

Recommendation

It is recommended to modify the code from *keccak256(abi.encodePacked(target, value, data))* to *keccak256(abi.encodePacked(permissions))* to ensure that parameters are appropriately checked within the specified range.

Patch Comment

We have confirmed that the leaf contains the hash of the Permission data, as we recommended. (https://github.com/zerodevapp/kernel/pull/22)

3. Wrong Parameter for _packValidationData upon Successful Merkle Proof in validateUserOp of SessionKeyValidator

ID: ZeroDevV2-1-03
Severity: Medium
Type: Logic Error
Difficulty: Low
File: src/validator/SessionKeyValidator.sol

Issue

```
function validateUserOp(UserOperation calldata userOp, bytes32 userOpHash, uint256 missingFunds)
        external
        payable
        returns (uint256)
{
    ...
    bytes32 leaf = keccak256(abi.encodePacked(target, value, data));
    bool result = MerkleProofLib.verify(merkleProof, session.merkleRoot, leaf);
    return _packValidationData(result, session.validUntil, session.validAfter);
}
```

https://github.com/zerodevapp/kernel/tree/199ae7d838f21b37f069267c86e8eeb6f0175a69/src/validator/SessionKeyValidator.sol

The SessionKeyValidator contract, designed to handle operations related to user session keys, contains the function validateUserOp. This function carries out the critical task of validating user operations through a Merkle Proof mechanism.
In the current implementation of the validateUserOp function, upon a successful Merkle Proof, the _packValidationData's first parameter is set to 1 (true). While 1(true) could intuitively signify a 'successful' operation, it has led to an unexpected problem due to how it is interpreted in the EntryPoint.
The EntryPoint interprets an input of 1 for _packValidationData as a failure of signature validation or SIG_VALIDATION_FAILED. This discrepancy between the 'success' value in the validateUserOp function and the way it's interpreted in the EntryPoint causes a conflict in the logic of the contract.

Recommendation

We recommended that the value set for _packValidationData upon a successful Merkle Proof in the validateUserOp function be changed to 0.

As the EntryPoint interprets 0 as a successful validation, this change would ensure that the validation process's success is correctly communicated and acknowledged across the entire system.

Patch Comment

We have verified that the first parameter of _packValidationData is perfectly compatible with EntryPoint, as we recommended. (https://github.com/zerodevapp/kernel/pull/22)

4. Wrong dataOffset Calculation in ValidateUserOp within SessionKeyValidator

ID: ZeroDevV2-1-04
Severity: Medium
Type: Logic Error
Difficulty: Low
File: src/validator/SessionKeyValidator.sol

Issue

The SessionKeyValidator smart contract's validateUserOp function plays a pivotal role in parsing the target, value, and data. This function is used when external functions are executed via the Kernel's execute function.

The callData layout used in the validateUserOp function is as follows:

[0:4] (4bytes)	Kernel Execute Function Sig
[4:36] (32bytes)	Executor address
[36:68] (32bytes)	value
[68:100] (32bytes)	data offset
[100:132] (32bytes)	data length
-	data

However, an issue arises when the Kernel Execute Function runs. The leading 4 bytes are removed, causing the subsequent fields to shift up by 4 bytes. During this shift, the data offset could be misread, potentially causing a revert event.

This incorrect interpretation of the data offset can have a significant impact on the overall functionality of the system. It may cause a revert event during the execution of the Kernel Execute Function, which could potentially halt further processing, disrupt service, and affect overall system integrity.

Recommendation

A change in how the data offset is parsed is recommended to address this. Instead of using the current line of code:

```
uint256 dataOffset = uint256(bytes32(userOp.callData[68:100]));
```

it should be replaced with:

```
uint256 dataOffset = uint256(bytes32(userOp.callData[68:100]))+4;
```

By adding +4, this change compensates for the 4-byte shift that occurs when the Kernel Execute Function runs, ensuring that the data offset is interpreted correctly and thus preventing the potential revert event.

Patch Comment

We have confirmed that 4 has been added to the dataOffset, as we recommended (https://github.com/zerodevapp/kernel/pull/22)

5. Discrepancies in Parsing validUntil and validAfter

ID: ZeroDevV2-1-05
Type: Logic Error
File: src/Kernel.sol

Severity: Medium
Difficulty: Low

Issue

The Kernel smart contract seems to be incorrectly parsing validUntil and validAfter. Two distinct approaches have been identified in the current code when parsing these values:

In bitwise operations, the order is below.

validAfter	>> 208	validUntil	>> 160

However, when dealing with arrays, the order is below.

validUntil	[4:10]	validAfter	[10:16]

This discrepancy has led to a situation where validUntil is less than validAfter, causing the UserOperation to result in a revert consistently.

The order of data in the UserOperation Signature is initially (validUntil, validAfter). so, this should be changed to (validAfter, validUntil).

Also, mode 0x00000001 and _approveValidator should be modified accordingly.

Recommendation

To rectify this issue, we suggest modifying the data ordering in the UserOperation Signature from validUntil, validAfter to validAfter, validUntil. The mode 0x00000001 and *_approveValidator* function should be updated to reflect this change. Specifically, the code in its bottom section should be modified as follows:

```
function validateUserOp(UserOperation calldata userOp, bytes32 userOpHash, uint256 missingAccountFunds)
    external
    payable
    returns (uint256 validationData)
```

```
{
    ...
    } else if (mode == 0x00000001) {
        bytes4 sig = bytes4(userOp.callData[0:4]);
        ExecutionDetail storage detail = kernelStorage.execution[sig];
        validator = detail.validator;
        if (address(validator) == address(0)) {
            validator = kernelStorage.defaultValidator;
        }
        op.signature = userOp.signature[4:];
        validationData = (uint256(detail.validAfter) << 208) | (uint256(detail.validUntil) << 160);
    }
    ...
}
...
function _approveValidator(bytes4 sig, bytes calldata signature)
    internal
    returns (uint256 validationData, bytes calldata enableData, bytes calldata validationSig)
{
    ...
    getKernelStorage().execution[sig] = ExecutionDetail({
        executor: address(bytes20(signature[36:56])),
        validAfter: uint48(bytes6(signature[4:10])),
        validUntil: uint48(bytes6(signature[10:16])),
        validator: IKernelValidator(address(bytes20(signature[16:36])))
    });
}
```

Patch Comment

We have verified that the parsing method has been changed as we recommended. (https://github.com/zerodevapp/kernel/commit/a0c14239aa38733eaae41a0d0a958a5000105a96)

6. Replay Attack in Setting DefaultValidator in KillSwitchValidator

ID: ZeroDevV2-1-06
Severity: Critical
Type: Logic Error
Difficulty: Low
File: src/validator/KillSwitchValidator.sol

Issue

The validateUserOp function in the KillSwitchValidator contract designates a Validator for validating UserOperation if the length of UserOperation's signature is 71. Although it verifies if the appointed Guardian has signed pausedUntil until it is vulnerable to replay attacks, a critical bug arises because the validator is determined by the data obtained from msg.sender.

Assuming a normal UserOperation calls the validateUserOp function, it would skip the first if statement due to the absence of a designated validator and proceed to the second if code to check if the UserOperation signature length is 71. Subsequently, a validator that is derived from msg.sender would be set.

```
function validateUserOp(UserOperation calldata _userOp, bytes32 _userOpHash, uint256)
        external
        payable
        override
        returns (uint256)
{
    KillSwitchValidatorStorage storage validatorStorage = killSwitchValidatorStorage[_userOp.sender];
// msg.sender
    uint48 pausedUntil = validatorStorage.pausedUntil;
    uint256 validationResult = 0;
    if (address(validatorStorage.validator) != address(0)) {
        // check for validator at first
        try validatorStorage.validator.validateUserOp(_userOp, _userOpHash, pausedUntil) returns
(uint256 res) { // ECDSAValidator
            validationResult = res;
        } catch {
            validationResult = SIG_VALIDATION_FAILED;
        }
        ValidationData memory validationData = _parseValidationData(validationResult);
        if (validationData.aggregator != address(1)) {
            // if signature verification has not been failed, return with the result
            uint256 delayedData = _packValidationData(false, 0, pausedUntil);
            return _packValidationData(_intersectTimeRange(validationResult, delayedData));
        }
    }
    if (_userOp.signature.length == 71) {
        // save data to this storage
        validatorStorage.pausedUntil = uint48(bytes6(_userOp.signature[0:6]));
        validatorStorage.validator = KernelStorage(msg.sender).getDefaultValidator();
        validatorStorage.disableMode = KernelStorage(msg.sender).getDisabledMode();
        bytes32 hash = ECDSA.toEthSignedMessageHash(keccak256(bytes.concat(_userOp.signature[0:6],
_userOpHash)));
```

```
        address recovered = ECDSA.recover(hash, _userOp.signature[6:]);
        if (validatorStorage.guardian != recovered) {
            return SIG_VALIDATION_FAILED;
        }
        return _packValidationData(false, 0, pausedUntil);
    } else {
        return SIG_VALIDATION_FAILED;
    }
}
```

https://github.com/zerodevapp/kernel/tree/199ae7d838f21b37f069267c86e8eeb6f0175a69/src/validator/KillSwitchValidator.sol

In a scenario where a malicious user reuses a previously used UserOperation to make a direct call to the validateUserOp function of the KillSwitchValidator contract bypassing the EntryPoint, the validator would attempt to verify if it's a valid UserOperation and fails. However, because there is neither a return statement nor a revert statement in case of failure, the function would continue to check if the UserOperation signature length is 71, and the validator would be reset by msg.sender (the malicious user). This mechanism could lead to a wallet takeover.

The root cause of this issue is the assignment of KillSwitchValidatorStorage based on _userOp.sender, while the validator is designated based on msg.sender.

This bug allows a malicious user to potentially usurp control over a wallet, leading to the loss of assets or potentially compromising the entire system's security. It poses a significant threat to the system's integrity and user trust.

Recommendation

To resolve this issue, a consistent approach should be employed for determining the base in both instances. Either KillSwitchValidatorStorage and validator should be based on _userOp.sender, or both should be based on msg.sender.

Patch Comment

We confirmed that the code for KillSwitchValidator has been modified as we recommended (https://github.com/zerodevapp/kernel/commit/e529b7978261e145c00e76833b7b95acf77e28c4).

7. Manipulation of defaultValidator and Execution Information in Session Leakage Scenario

ID: ZeroDevV2-1-07
Severity: High
Type: Logic Error
Difficulty: Low
File: src/Kernel.sol

Issue

The defaultValidator should only be set through the ECDSAValidator designated during the Kernel creation.
The defaultValidator has the authority to arbitrarily activate a validator to verify the validity of a UserOperation for a specific action. If it is arbitrarily altered, a malicious attacker can seize wallet ownership, which would be impossible to recover.
Assuming a session key leakage, this situation could be highly critical.
Suppose a session key that can call the execute function of the Kernel contract is leaked. In that case, it is possible to run the setExecution and setDefaultValidator functions via a self-call through the Kernel contract's execute function.
This vulnerability allows for the potential manipulation of the defaultValidator, and in the event of a session key leak, it presents a severe security risk.
An attacker could steal wallet ownership, and the loss would be irreversible. It can severely impact user trust and the integrity of the system.

Recommendation

By activating the SessionKeyValidator, the Wallet Owner must manage it properly to ensure that a Session with limited permissions cannot execute functions in the Kernel Contract that may cause issues related to Kernel ownership, mainly functions like upgradeTo, setExecution, and setDefaultValidator. Moreover, considering scenarios where the Session Key might be compromised, it cannot be ruled out that it might be granted excessive permissions.
Therefore, we recommend modifying the code to prevent the Merkle Root from being set to 0, ensuring it does not have undue authority.

Patch Comment

(TBD)

8. DelegateCall Risk

ID: ZeroDevV2-1-08
Type: Logic Error
File: src/Kernel.sol

Severity: Tips
Difficulty: -

Issue

Misusing the DelegateCall option could lead to overwriting crucial storage variables due to Storage Layout Collisions, leading to unpredictable results. Such an issue could lead to data corruption or loss, severely impacting the functioning of the contract and possibly the overall system.

```
function execute(address to, uint256 value, bytes calldata data, Operation operation) external payable
{
    if (msg.sender != address(entryPoint) && !_checkCaller()) {
        revert NotAuthorizedCaller();
    }
    bytes memory callData = data;
    if (operation == Operation.DelegateCall) {
        assembly {
            let success := delegatecall(gas(), to, add(callData, 0x20), mload(callData), 0, 0)
            returndatacopy(0, 0, returndatasize())
            switch success
            case 0 { revert(0, returndatasize()) }
            default { return(0, returndatasize()) }
        }
    } else {
        assembly {
            let success := call(gas(), to, value, add(callData, 0x20), mload(callData), 0, 0)
            returndatacopy(0, 0, returndatasize())
            switch success
            case 0 { revert(0, returndatasize()) }
            default { return(0, returndatasize()) }
        }
    }
}
```

https://github.com/zerodevapp/kernel/tree/199ae7d838f21b37f069267c86e8eeb6f0175a69/src/Kernel.sol

The current design of the Kernel contract's execute function includes a DelegateCall option. This DelegateCall option, as per the analysis of the ZeroDev SDK, only seems to be used for the Multicall feature and does not have any other function. However, if DelegateCall is misused, it could lead to a Storage Layout Collision. This issue could cause vital storage variables to be overwritten and trigger unexpected outcomes.

Recommendation

Similar to issue #7, if the Session Key is compromised and the SessionKeyValidator is activated with a lax Permission Rule, there is a possibility of a Storage Layout Collision due to

DelegateCall. To mitigate this risk, we recommend adding a check for Call/DelegateCall in the Permission Rule that the SessionKeyValidator examines.

Patch Comment

We have confirmed that the code for SessionKeyValidator has been modified as we recommended (https://github.com/zerodevapp/kernel/commit/adc0513e2d5695daa38b345665540d4af25745ac).

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