// Smart Contract Security Assessment 12.23.2024 - 01.03.2025

Account Abstraction Schnorr MultiSig

InFlux Technologies

HALBORN

Account Abstraction Schnorr MultiSig - InFlux Technologies

Prepared by: HALBORN

Last Updated 01/10/2025

Date of Engagement by: December 23rd, 2024 - January 3rd, 2025

Summary

100% ⓘ OF ALL REPORTED FINDINGS HAVE BEEN ADDRESSED

ALL FINDINGS	CRITICAL	HIGH	MEDIUM	LOW	INFORMATIONAL
5	0	0	0	2	3

TABLE OF CONTENTS

1. Introduction

`InFlux Technologies` engaged `Halborn` to conduct a security assessment on their smart contracts revisions started on December 23th, 2024 and ending on January 3rd, 2025. The security assessment was scoped to the smart contracts provided to the `Halborn` team.

Commit hashes and further details can be found in the Scope section of this report.

2. Assessment Summary

The team at Halborn was provided 1 week and 2 days for the engagement and assigned a security engineer to evaluate the security of the smart contract.

The security engineer is a blockchain and smart-contract security expert with advanced penetration testing, smart-contract hacking, and deep knowledge of multiple blockchain protocols.

The purpose of this assessment is to:

- Ensure that smart contract functions operate as intended.
- Identify potential security issues with the smart contracts.

In summary, Halborn identified some improvements to reduce the likelihood and impact of risks, which were mostly acknowledged by the `InFlux Technologies team`. The main ones were the following:

- `Add some checks for invalid time range.`
- `Limit message length return.`
- `Add some sanity checks.`

3. Test Approach And Methodology

Halborn performed a combination of manual and automated security testing to balance efficiency, timeliness, practicality, and accuracy regarding the scope of this assessment. While manual testing is recommended to uncover flaws in logic, process, and implementation; automated testing techniques help enhance code coverage and quickly identify items that do not follow the security best practices. The following phases and associated tools were used during the assessment:

- Research into architecture and purpose.
- Smart contract manual code review and walkthrough.
- Graphing out functionality and contract logic/connectivity/functions. (`solgraph`,`draw.io`)
- Manual assessment of use and safety for the critical Solidity variables and functions in scope to identify any arithmetic related vulnerability classes.
- Manual testing by custom scripts.
- Static Analysis of security for scoped contract, and imported functions. (`Slither`)
- Testnet deployment. (`Hardhat`,`Foundry`)

4. RISK METHODOLOGY

Every vulnerability and issue observed by Halborn is ranked based on **two sets** of **Metrics** and a **Severity Coefficient**. This system is inspired by the industry standard Common Vulnerability Scoring System.

The two **Metric sets** are: **Exploitability** and **Impact**. **Exploitability** captures the ease and technical means by which vulnerabilities can be exploited and **Impact** describes the consequences of a successful exploit.

The **Severity Coefficients** is designed to further refine the accuracy of the ranking with two factors: **Reversibility** and **Scope**. These capture the impact of the vulnerability on the environment as well as the number of users and smart contracts affected.

The final score is a value between 0-10 rounded up to 1 decimal place and 10 corresponding to the highest security risk. This provides an objective and accurate rating of the severity of security vulnerabilities in smart contracts.

The system is designed to assist in identifying and prioritizing vulnerabilities based on their level of risk to address the most critical issues in a timely manner.

4.1 EXPLOITABILITY

ATTACK ORIGIN (AO):

Captures whether the attack requires compromising a specific account.

ATTACK COST (AC):

Captures the cost of exploiting the vulnerability incurred by the attacker relative to sending a single transaction on the relevant blockchain. Includes but is not limited to financial and computational cost.

ATTACK COMPLEXITY (AX):

Describes the conditions beyond the attacker's control that must exist in order to exploit the vulnerability. Includes but is not limited to macro situation, available third-party liquidity and regulatory challenges.

METRICS:

EXPLOITABILITY METRIC (M_E)	METRIC VALUE	NUMERICAL VALUE
Attack Origin (AO)	Arbitrary (AO:A) Specific (AO:S)	1 0.2
Attack Cost (AC)	Low (AC:L) Medium (AC:M) High (AC:H)	1 0.67 0.33

EXPLOITABILITY METRIC (M_E)	METRIC VALUE	NUMERICAL VALUE
Attack Complexity (AX)	Low (AX:L) Medium (AX:M) High (AX:H)	1 0.67 0.33

Exploitability E is calculated using the following formula:

$$E = \prod m_e$$

4.2 IMPACT

CONFIDENTIALITY (C):

Measures the impact to the confidentiality of the information resources managed by the contract due to a successfully exploited vulnerability. Confidentiality refers to limiting access to authorized users only.

INTEGRITY (I):

Measures the impact to integrity of a successfully exploited vulnerability. Integrity refers to the trustworthiness and veracity of data stored and/or processed on-chain. Integrity impact directly affecting Deposit or Yield records is excluded.

AVAILABILITY (A):

Measures the impact to the availability of the impacted component resulting from a successfully exploited vulnerability. This metric refers to smart contract features and functionality, not state. Availability impact directly affecting Deposit or Yield is excluded.

DEPOSIT (D):

Measures the impact to the deposits made to the contract by either users or owners.

YIELD (Y):

Measures the impact to the yield generated by the contract for either users or owners.

METRICS:

IMPACT METRIC (M_I)	METRIC VALUE	NUMERICAL VALUE
Confidentiality (C)	None (I:N) Low (I:L) Medium (I:M) High (I:H) Critical (I:C)	0 0.25 0.5 0.75 1

IMPACT METRIC (M_I)	METRIC VALUE	NUMERICAL VALUE
Integrity (I)	None (I:N) Low (I:L) Medium (I:M) High (I:H) Critical (I:C)	0 0.25 0.5 0.75 1
Availability (A)	None (A:N) Low (A:L) Medium (A:M) High (A:H) Critical (A:C)	0 0.25 0.5 0.75 1
Deposit (D)	None (D:N) Low (D:L) Medium (D:M) High (D:H) Critical (D:C)	0 0.25 0.5 0.75 1
Yield (Y)	None (Y:N) Low (Y:L) Medium (Y:M) High (Y:H) Critical (Y:C)	0 0.25 0.5 0.75 1

Impact I is calculated using the following formula:

$$I = max(m_I) + \frac{\sum m_I - max(m_I)}{4}$$

4.3 SEVERITY COEFFICIENT

REVERSIBILITY (R):

Describes the share of the exploited vulnerability effects that can be reversed. For upgradeable contracts, assume the contract private key is available.

SCOPE (S):

Captures whether a vulnerability in one vulnerable contract impacts resources in other contracts.

METRICS:

SEVERITY COEFFICIENT (C)	COEFFICIENT VALUE	NUMERICAL VALUE
Reversibility (r)	None (R:N) Partial (R:P) Full (R:F)	1 0.5 0.25
Scope (s)	Changed (S:C) Unchanged (S:U)	1.25 1

Severity Coefficient C is obtained by the following product:

$$C = rs$$

The Vulnerability Severity Score S is obtained by:

$$S = min(10, EIC * 10)$$

The score is rounded up to 1 decimal places.

SEVERITY	SCORE VALUE RANGE
Critical	9 - 10
High	7 - 8.9
Medium	4.5 - 6.9
Low	2 - 4.4
Informational	0 - 1.9

5. SCOPE

FILES AND REPOSITORY

(a) Repository: account-abstraction

(b) Assessed Commit ID: 588c582

(c) Items in scope:

- IMultiSigSmartAccount
- IMultiSigSmartAccountFactory
- interfaces/INonceManager
- interfaces/IUserOperation
- interfaces/IStakeManager
- interfaces/IAggregator
- interfaces/IPaymaster
- interfaces/IEntryPoint
- interfaces/IAccount
- utils/Helpers
- utils/Exec
- utils/TokenCallbackHandler
- core/EntryPoint
- core/BaseAccount
- core/SenderCreator
- core/UserOperation
- core/NonceManager
- core/StakeManager
- core/BasePaymaster
- Schnorr
- Schnorr
- MultiSigSmartAccountFactory
- MultiSigSmartAccount

Out-of-Scope: Third Party Dependencies., Economic Attacks.

Out-of-Scope: New features/implementations after the remediation commit IDs.

6. ASSESSMENT SUMMARY & FINDINGS OVERVIEW

CRITICAL	HIGH	MEDIUM	LOW	INFORMATIONAL

0 0 0 2 3

SECURITY ANALYSIS	RISK LEVEL	REMEDIATION DATE
INVALID TIME RANGE CAN BE RETURNED BY _INTERSECTTIMERANGE	LOW	RISK ACCEPTED - 01/08/2025
UNBOUNDED ERROR MESSAGE LENGTH IN PAYMASTER EXTERNAL CALLS CREATES GAS UNPREDICTABILITY	LOW	NOT APPLICABLE
MISSING SANITY CHECK FOR ENTRYPOINT	INFORMATIONAL	ACKNOWLEDGED - 01/08/2025
LACK OF ACCOUNT DEPLOYMENT VERIFICATION CAN LEAD TO UNEXPECTED REVERT BEHAVIOR	INFORMATIONAL	ACKNOWLEDGED - 01/08/2025
OVERLY BROAD UNCHECKED MATH BLOCKS IN ENTRYPOINT CONTRACT	INFORMATIONAL	ACKNOWLEDGED - 01/08/2025

7. FINDINGS & TECH DETAILS

7.1 INVALID TIME RANGE CAN BE RETURNED BY _INTERSECTTIMERANGE

// LOW

Description

The `_intersectTimeRange` function in the Helper library contains a logical flaw in its time range validation mechanism. When intersecting time ranges between account and paymaster validations, the function fails to verify if the resulting time window is valid. The issue occurs in the following sequence:

- The function takes the maximum value between account's and paymaster's `validAfter`
- The function takes the minimum value between account's and paymaster's `validUntil`
- No validation exists to ensure final `validAfter <= validUntil`

```
37      // intersect account and paymaster ranges.
38      function _intersectTimeRange(
39          uint256 validationData,
40          uint256 paymasterValidationData
41      ) internal pure returns (ValidationData memory) {
42          ValidationData memory accountValidationData = _parseValidationD
43          ValidationData memory pmValidationData = _parseValidationData(p
44          address aggregator = accountValidationData.aggregator;
45          if (aggregator == address(0)) {
46              aggregator = pmValidationData.aggregator;
47          }
48          uint48 validAfter = accountValidationData.validAfter;
49          uint48 validUntil = accountValidationData.validUntil;
50          uint48 pmValidAfter = pmValidationData.validAfter;
51          uint48 pmValidUntil = pmValidationData.validUntil;
52
53          if (validAfter < pmValidAfter) validAfter = pmValidAfter;
54          if (validUntil > pmValidUntil) validUntil = pmValidUntil;
55
56          // @tocheck missing validation to ensure validAfter <= validUnt
57
58          return ValidationData(aggregator, validAfter, validUntil);
59      }
```

It could result in :

- Invalid time ranges being accepted by the system
- Broken time-based validation mechanisms

Proof of Concept

This POC can be reproduced using this code :

```
import { expect } from "chai";
import { ethers } from "hardhat";
import { HelperTest } from "../contracts/HelperTest.sol"; // Adjust import p

describe("Helper", () => {
  let helperTest: HelperTest;

  beforeEach(async () => {
    const Helper = await ethers.getContractFactory("HelperTest");
    helperTest = await Helper.deploy();
  });

  it("should demonstrate validAfter > validUntil in _intersectTimeRange", as
    // Create validation data where account is valid from t=100 to t=200
    const accountValidationData = helperTest.packValidationData({
      aggregator: ethers.ZeroAddress,
      validAfter: 100n,
      validUntil: 200n
    });

    // Create paymaster validation data where paymaster is valid from t=150
    const paymasterValidationData = helperTest.packValidationData({
      aggregator: ethers.ZeroAddress,
      validAfter: 150n,
      validUntil: 120n
    });

    // Intersect the time ranges
    const result = await helperTest.intersectTimeRange(
      accountValidationData,
      paymasterValidationData
    );

    const validAfter = Number(result.validAfter);
    const validUntil = Number(result.validUntil);

    // validAfter (150) is greater than validUntil (120)
    expect(validAfter).to.be.greaterThan(validUntil);
    expect(validAfter).to.equal(150);
    expect(validUntil).to.equal(120);

    console.log(`ValidAfter: ${validAfter}, ValidUntil: ${validUntil}`);
```

```
  });
});
```

Result :

```
account-abstraction git:(master) ✗ npx hardhat test --grep "Helper"

  Helper
ValidAfter: 150, ValidUntil: 120
    ✔ should demonstrate validAfter > validUntil in _intersectTimeRange

  1 passing (979ms)
```

BVSS

AO:A/AC:L/AX:L/C:N/I:L/A:N/D:N/Y:N/R:N/S:U (2.5)

Recommendation

It is recommended to add another validation check after the time range intersection and mark the validation as failed when an invalid range is detected.

Remediation

RISK ACCEPTED: An updated will be used for new chains. However, **Influx** will be using the current deployed on chain version for chains where is it possible.

References

RunOnFlux/account-abstraction/contracts/erc4337/utils/Helpers.sol#L38

7.2 UNBOUNDED ERROR MESSAGE LENGTH IN PAYMASTER EXTERNAL CALLS CREATES GAS UNPREDICTABILITY

// LOW

Description

The `EntryPoint` contract handles error messages from paymaster calls without imposing size limits. This occurs in two critical locations where external calls to paymaster contracts are made: `_validatePaymasterPrepayment` and `_handlePostOp`. First instance in `_validatePaymasterPrepayment`:

```
476 function _validatePaymasterPrepayment(
477     uint256 opIndex,
478     UserOperation calldata op,
479     UserOpInfo memory opInfo,
480     uint256 requiredPreFund,
481     uint256 gasUsedByValidateAccountPrepayment
482 ) internal returns (bytes memory context, uint256 validationData) {
483     // .... //
484     try IPaymaster(paymaster).validatePaymasterUserOp{gas: gas}(op, opI
485         returns (bytes memory _context, uint256 _validationData) {
486             context = _context;
487             validationData = _validationData;
488     } catch Error(string memory revertReason) {
489         revert FailedOp(opIndex, string.concat("AA33 reverted: ", rever
490     }
491 }
```

Second instance in `_handlePostOp`:

```
625 function _handlePostOp(
626     uint256 opIndex,
627     IPaymaster.PostOpMode mode,
628     UserOpInfo memory opInfo,
629     bytes memory context,
630     uint256 actualGas
631 ) private returns (uint256 actualGasCost) {
632     // ... //
633     try IPaymaster(paymaster).postOp{gas: mUserOp.verificationGasLimit}
634     {} catch Error(string memory reason) {
635         revert FailedOp(opIndex, string.concat("AA50 postOp reverted: "
636
637
```

```
        }
    }
```

While the contract defines `REVERT_REASON_MAX_LEN = 2048`, this limit is not enforced when handling these paymaster error messages.
Report as informational as it's paymaster handling but there are several impacts here :

- Gas consumption becomes unpredictable when copying large error messages to memory
- Malicious paymasters can force excessive gas consumption through large error messages
- Bundle execution can fail unexpectedly due to out-of-gas conditions

BVSS

AO:A/AC:L/AX:L/C:N/I:N/A:L/D:N/Y:N/R:N/S:U (2.5)

Recommendation

It is recommended to implement error message length limits using the existing `REVERT_REASON_MAX_LEN` constant.

Remediation

NOT APPLICABLE: Non concerning because the current deployed files does not use the affected code.

References

RunOnFlux/account-abstraction/contracts/erc4337/core/EntryPoint.sol#L625
RunOnFlux/account-abstraction/contracts/erc4337/core/EntryPoint.sol#L476

7.3 MISSING SANITY CHECK FOR ENTRYPOINT

// INFORMATIONAL

Description

The `BasePaymaster` contract lacks interface validation for the _entryPoint parameter in its constructor. The current implementation directly sets the `EntryPoint` address without verifying if it implements a matching `IEntryPoint` interface.

```
19  constructor(IEntryPoint _entryPoint, address _owner) Ownable(_owner) {
20
21      //E @tocheck missing sanity check for _entryPoint
22
23      setEntryPoint(_entryPoint);
24  }
```

Score

AO:A/AC:L/AX:L/C:N/I:N/A:N/D:N/Y:N/R:N/S:U (0.0)

Recommendation

It is recommended to implement the same mechanism as it is done in eth-infinitism development branch repository which includes proper validation with an interface to be implemented:

```
constructor(IEntryPoint _entryPoint) Ownable(msg.sender) {

    _validateEntryPointInterface(_entryPoint);

    entryPoint = _entryPoint;

}

function _validateEntryPointInterface(IEntryPoint _entryPoint) internal vir
   require(IERC165(address(_entryPoint)).supportsInterface(type(IEntryPoint
}
```

Remediation

ACKNOWLEDGED: An on chain version of this file will be used instead of the one in the repository, but the project does not use the affected code.

References

RunOnFlux/account-abstraction/contracts/erc4337/core/BasePaymaster.sol#L19

7.4 LACK OF ACCOUNT DEPLOYMENT VERIFICATION CAN LEAD TO UNEXPECTED REVERT BEHAVIOR

// INFORMATIONAL

Description

The `_validateAccountPrepayment` function in EntryPoint.sol fails to verify the existence of code at the sender's address when `initCode` is empty. While the function deploys new accounts when `initCode` is provided, it assumes pre-existing accounts are already deployed without verification.

```
426 function _validateAccountPrepayment(
427         uint256 opIndex,
428         UserOperation calldata op,
429         UserOpInfo memory opInfo,
430         uint256 requiredPrefund
431 ) internal returns (uint256 gasUsedByValidateAccountPrepayment, uint256
432         unchecked {
433             MemoryUserOp memory mUserOp = opInfo.mUserOp;
434             address sender = mUserOp.sender;
435             _createSenderIfNeeded(opIndex, opInfo, op.initCode);
436             // Missing verification of sender.code.length
437             try IAccount(sender).validateUserOp{gas: mUserOp.verificati
438             // ... //
```

The issue occurs because the subsequent call to `validateUserOp` will revert at the EVM level if no code exists at the sender's address. This revert happens before entering the `try-catch` block, preventing the emission of the expected `FailedOp` error.

The risk is that `Bundlers` receive unclear error feedback when processing transactions for non-deployed accounts.

Score

AO:A/AC:L/AX:L/C:N/I:N/A:N/D:N/Y:N/R:N/S:C (0.0)

Recommendation

It is recommended to add an explicit deployment check after `_createSenderIfNeeded` to ensure consistent error handling and provide clear feedback about account deployment status.

Remediation

ACKNOWLEDGED: An on chain version of this file will be used instead of the one in the repository, but the project does not use the affected code.

References

RunOnFlux/account-abstraction/contracts/erc4337/core/EntryPoint.sol#L426

7.5 OVERLY BROAD UNCHECKED MATH BLOCKS IN ENTRYPOINT CONTRACT

// INFORMATIONAL

Description

The EntryPoint contract contains multiple functions where entire function bodies are wrapped in unchecked blocks, rather than limiting these blocks to specific arithmetic operations that require them. This pattern is observed in critical functions including `_validateAccountPrepayment`, `_validatePaymasterPrepayment`, and `_execute` instances:

```
426 function _validateAccountPrepayment(
427     uint256 opIndex,
428     UserOperation calldata op,
429     UserOpInfo memory opInfo,
430     uint256 requiredPrefund
431 ) internal returns (uint256 gasUsedByValidateAccountPrepayment, uint256
432
433     unchecked {
434         // Entire function body (50+ lines) within unchecked block
435         uint256 preGas = gasleft();
436         // ... many operations that don't require unchecked math
437         gasUsedByValidateAccountPrepayment = preGas - gasleft();
438     }
439
440 }
```

```
476 function _validatePaymasterPrepayment(
477     uint256 opIndex,
478     UserOperation calldata op,
479     UserOpInfo memory opInfo,
480     uint256 requiredPreFund,
481     uint256 gasUsedByValidateAccountPrepayment
482 ) internal returns (bytes memory context, uint256 validationData) {
483
484     unchecked {
485         // Entire function body within unchecked block
486         // ... including operations that don't require unchecked math
487         paymasterInfo.deposit = uint112(deposit - requiredPreFund);
488     }
489
490 }
```

Impacts:

- Reduced code clarity and increased review complexity
- Higher risk of arithmetic overflow/underflow vulnerabilities

Score

AO:A/AC:L/AX:L/C:N/I:N/A:N/D:N/Y:N/R:N/S:C (0.0)

Recommendation

It is recommended to restrict unchecked blocks to specific arithmetic operations
where overflow/underflow checks are unnecessary.

Remediation

ACKNOWLEDGED: An on chain version of this file will be used instead of the one in the repository, but the project does not use the affected code.

References

RunOnFlux/account-abstraction/contracts/erc4337/core/EntryPoint.sol#L426
RunOnFlux/account-abstraction/contracts/erc4337/core/EntryPoint.sol#L476

8. AUTOMATED TESTING

Halborn used automated testing techniques to enhance the coverage of certain areas of the smart contracts in scope. Among the tools used was `Slither`, a Solidity static analysis framework.
After Halborn verified the smart contracts in the repository and was able to compile them correctly into their abis and binary format, Slither was run against the contracts. This tool can statically verify mathematical relationships between Solidity variables to detect invalid or inconsistent usage of the contracts' APIs across the entire code-base.

```
→  account-abstraction git:(master) ✗ slither .  --exclude-informational --exclude-low
'npx hardhat clean' running (wd: /Users/liliancariou/Desktop/Halborn/audits/account-abstraction)
'npx hardhat clean --global' running (wd: /Users/liliancariou/Desktop/Halborn/audits/account-abstraction)
'npx hardhat compile --force' running (wd: /Users/liliancariou/Desktop/Halborn/audits/account-abstraction)
INFO:Detectors:
MultiSigSmartAccount._call(address,uint256,bytes) (contracts/MultiSigSmartAccount.sol#150-157) sends eth to arbitrary user
        Dangerous calls:
        - (success,result) = target.call{value: value}(data) (contracts/MultiSigSmartAccount.sol#151)
BaseAccount._payPrefund(uint256) (contracts/erc4337/core/BaseAccount.sol#75-82) sends eth to arbitrary user
        Dangerous calls:
        - (success) = address(msg.sender).call{gas: type()(uint256).max,value: missingAccountFunds}() (contracts/erc4337/core/BaseAccount.sol#79)
EntryPoint._compensate(address,uint256) (contracts/erc4337/core/EntryPoint.sol#393-397) sends eth to arbitrary user
        Dangerous calls:
        - (success) = beneficiary.call{value: amount}() (contracts/erc4337/core/EntryPoint.sol#395)
Reference: https://github.com/crytic/slither/wiki/Detector-Documentation#functions-that-send-ether-to-arbitrary-destinations
INFO:Detectors:
Math.mulDiv(uint256,uint256,uint256) (node_modules/@openzeppelin/contracts/utils/math/Math.sol#123-202) has bitwise-xor operator ^ instead of the exponentiation operator **:
         - inverse = (3 * denominator) ^ 2 (node_modules/@openzeppelin/contracts/utils/math/Math.sol#184)
Reference: https://github.com/crytic/slither/wiki/Detector-Documentation#incorrect-exponentiation
INFO:Detectors:
IAccount is re-used:
        - IAccount (contracts/erc4337/interfaces/IAccount.sol#6-36)
        - IAccount (contracts/erc4337/interfaces/IUserOperation.sol#6-36)
Reference: https://github.com/crytic/slither/wiki/Detector-Documentation#name-reused
```

```
INFO:Detectors:
EntryPoint.simulateHandleOp(UserOperation,address,bytes).targetSuccess (contracts/erc4337/core/EntryPoint.sol#177) is a local variable never initialized
EntryPoint._validatePrepayment(uint256,UserOperation,EntryPoint.UserOpInfo).context (contracts/erc4337/core/EntryPoint.sol#691) is a local variable never initialized
EntryPoint.simulateHandleOp(UserOperation,address,bytes).opInfo (contracts/erc4337/core/EntryPoint.sol#148) is a local variable never initialized
EntryPoint.simulateHandleOp(UserOperation,address,bytes).targetResult (contracts/erc4337/core/EntryPoint.sol#178) is a local variable never initialized
EntryPoint.simulateValidation(UserOperation).outOpInfo (contracts/erc4337/core/EntryPoint.sol#80) is a local variable never initialized
Reference: https://github.com/crytic/slither/wiki/Detector-Documentation#uninitialized-local-variables
INFO:Detectors:
ERC1967Utils.upgradeToAndCall(address,bytes) (node_modules/@openzeppelin/contracts/proxy/ERC1967/ERC1967Utils.sol#83-92) ignores return value by Address.functionDelegateCall(newI
mplementation,data) (node_modules/@openzeppelin/contracts/proxy/ERC1967/ERC1967Utils.sol#88)
ERC1967Utils.upgradeBeaconToAndCall(address,bytes) (node_modules/@openzeppelin/contracts/proxy/ERC1967/ERC1967Utils.sol#173-182) ignores return value by Address.functionDelegateC
all(IBeacon(newBeacon).implementation(),data) (node_modules/@openzeppelin/contracts/proxy/ERC1967/ERC1967Utils.sol#178)
Reference: https://github.com/crytic/slither/wiki/Detector-Documentation#unused-return
INFO:Slither:. analyzed (54 contracts with 58 detectors), 25 result(s) found
```

```
INFO:Detectors:
Reentrancy in EntryPoint._validateAccountPrepayment(uint256,UserOperation,EntryPoint.UserOpInfo,uint256) (contracts/erc4337/core/EntryPoint.sol#734-792):
        External calls:
        - _createSenderIfNeeded(opIndex,opInfo,op.initCode) (contracts/erc4337/core/EntryPoint.sol#750)
                - sender1 = senderCreator.createSender{gas: opInfo.mUserOp.verificationGasLimit}(initCode) (contracts/erc4337/core/EntryPoint.sol#588)
        - _validationData = IAccount(sender).validateUserOp{gas: mUserOp.verificationGasLimit}(op,opInfo.userOpHash,missingAccountFunds) (contracts/erc4337/core/EntryPoint.sol#77
1-777)
        State variables written after the call(s):
        - senderInfo.deposit = uint112(deposit - requiredPrefund) (contracts/erc4337/core/EntryPoint.sol#786)
        StakeManager.deposits (contracts/erc4337/core/StakeManager.sol#30) can be used in cross function reentrancies:
        - StakeManager._getStakeInfo(address) (contracts/erc4337/core/StakeManager.sol#39-43)
        - StakeManager._incrementDeposit(address,uint256) (contracts/erc4337/core/StakeManager.sol#66-71)
        - EntryPoint._validateAccountPrepayment(uint256,UserOperation,EntryPoint.UserOpInfo,uint256) (contracts/erc4337/core/EntryPoint.sol#734-792)
        - EntryPoint._validatePaymasterPrepayment(uint256,UserOperation,EntryPoint.UserOpInfo,uint256,uint256) (contracts/erc4337/core/EntryPoint.sol#798-838)
        - StakeManager.addStake(uint32) (contracts/erc4337/core/StakeManager.sol#75-93)
        - StakeManager.balanceOf(address) (contracts/erc4337/core/StakeManager.sol#46-48)
        - StakeManager.depositTo(address) (contracts/erc4337/core/StakeManager.sol#57-64)
        - StakeManager.deposits (contracts/erc4337/core/StakeManager.sol#30)
        - StakeManager.getDepositInfo(address) (contracts/erc4337/core/StakeManager.sol#33-35)
        - StakeManager.unlockStake() (contracts/erc4337/core/StakeManager.sol#96-108)
        - StakeManager.withdrawStake(address) (contracts/erc4337/core/StakeManager.sol#111-131)
        - StakeManager.withdrawTo(address,uint256) (contracts/erc4337/core/StakeManager.sol#134-146)
Reentrancy in EntryPoint._validatePrepayment(uint256,UserOperation,EntryPoint.UserOpInfo) (contracts/erc4337/core/EntryPoint.sol#641-710):
        External calls:
        - (gasUsedByValidateAccountPrepayment,validationData) = _validateAccountPrepayment(opIndex,userOp,outOpInfo,requiredPreFund) (contracts/erc4337/core/EntryPoint.sol#681)
                - sender1 = senderCreator.createSender{gas: opInfo.mUserOp.verificationGasLimit}(initCode) (contracts/erc4337/core/EntryPoint.sol#588)
                - _validationData = IAccount(sender).validateUserOp{gas: mUserOp.verificationGasLimit}(op,opInfo.userOpHash,missingAccountFunds) (contracts/erc4337/core/EntryPoin
t.sol#771-777)
        - (context,paymasterValidationData) = _validatePaymasterPrepayment(opIndex,userOp,outOpInfo,requiredPreFund,gasUsedByValidateAccountPrepayment) (contracts/erc4337/core/En
tryPoint.sol#695)
                - (_context,_validationData) = IPaymaster(paymaster).validatePaymasterUserOp{gas: gas}(op,opInfo.userOpHash,requiredPreFund) (contracts/erc4337/core/EntryPoint.so
l#825-836)
```

All issues identified by `Slither` were proved to be false positives or have been added to the issue list in this report.

Halborn strongly recommends conducting a follow-up assessment of the project either within six months or immediately following any material changes to the codebase, whichever comes first. This approach is crucial for maintaining the project's integrity and addressing potential vulnerabilities introduced by code modifications.