ackee blockchain security

Safe

ERC-4337 Module

by Ackee Blockchain

5.12.2023

Contents

1. Document Revisions

0.1	Draft report	3.11.2023
1.1	Final report	6.11.2023
1.1	Fix review	8.11.2023
1.2	Fix review of internal findings	27.11.2023
2.0	Final report version 2.0	5.12.2023

2. Overview

This document presents our findings in reviewed contracts.

2.1. Ackee Blockchain

Ackee Blockchain is an auditing company based in Prague, Czech Republic, specializing in audits and security assessments. Our mission is to build a stronger blockchain community by sharing knowledge – we run free certification courses School of Solana, Summer School of Solidity and teach at the Czech Technical University in Prague. Ackee Blockchain is backed by the largest VC fund focused on blockchain and DeFi in Europe, RockawayX.

2.2. Audit Methodology

1. **Technical specification/documentation** - a brief overview of the system is requested from the client and the scope of the audit is defined.
2. **Tool-based analysis** - deep check with automated Solidity analysis tools and Wake is performed.
3. **Manual code review** - the code is checked line by line for common vulnerabilities, code duplication, best practices and the code architecture is reviewed.
4. **Local deployment + hacking** - the contracts are deployed locally and we try to attack the system and break it.
5. **Unit and fuzz testing** - run unit tests to ensure that the system works as expected, potentially write missing unit or fuzz tests.

2.3. Finding classification

A *Severity* rating of each finding is determined as a synthesis of two sub-ratings: *Impact* and *Likelihood*. It ranges from *Informational* to *Critical*.

If we have found a scenario in which an issue is exploitable, it will be assigned an impact rating of *High*, *Medium*, or *Low*, based on the direness of the consequences it has on the system. If we haven't found a way, or the issue is only exploitable given a change in configuration (such as deployment scripts, compiler configuration, use of multi-signature wallets for owners, etc.) or given a change in the codebase, then it will be assigned an impact rating of *Warning* or *Info*.

Low to *High* impact issues also have a *Likelihood*, which measures the probability of exploitability during runtime.

The full definitions are as follows:

Severity

		Likelihood			
		High	**Medium**	**Low**	**-**
Impact	**High**	Critical	High	Medium	-
	Medium	High	Medium	Low	-
	Low	Medium	Low	Low	-
	Warning	-	-	-	Warning
	Info	-	-	-	Info

Table 1. Severity of findings

Impact

- **High** - Code that activates the issue will lead to undefined or catastrophic consequences for the system.
- **Medium** - Code that activates the issue will result in consequences of serious substance.
- **Low** - Code that activates the issue will have outcomes on the system that are either recoverable or don't jeopardize its regular functioning.
- **Warning** - The issue cannot be exploited given the current code and/or configuration (such as deployment scripts, compiler configuration, use of multi-signature wallets for owners, etc.), but could be a security vulnerability if these were to change slightly. If we haven't found a way to exploit the issue given the time constraints, it might be marked as a "Warning" or higher, based on our best estimate of whether it is currently exploitable.
- **Info** - The issue is on the borderline between code quality and security. Examples include insufficient logging for critical operations. Another example is that the issue would be security-related if code or configuration (see above) was to change.

Likelihood

- **High** - The issue is exploitable by virtually anyone under virtually any circumstance.
- **Medium** - Exploiting the issue currently requires non-trivial preconditions.
- **Low** - Exploiting the issue requires strict preconditions.

2.4. Review team

Member's Name	Position
Lukáš Böhm	Lead Auditor
Jan Kalivoda	Auditor
Josef Gattermayer, Ph.D.	Audit Supervisor

2.5. Disclaimer

We've put our best effort to find all vulnerabilities in the system, however our findings shouldn't be considered as a complete list of all existing issues. The statements made in this document should not be interpreted as investment or legal advice, nor should its authors be held accountable for decisions made based on them.

3. Executive Summary

The repository provided by Safe contains two smart contracts: `Safe4337Module` and `AddModulesLib`. The codebase extends the functionality of the Safe account wallet with a new contract, which allows ERC-4337 compatibility.

Revision 1.0

Safe engaged Ackee Blockchain to perform a security review of the Safe protocol with a total time donation of 3 engineering days in a period between October 30 and November 3, 2023 and the lead auditor was Lukáš Böhm. The audit has been performed on the commit `53211cc` [1] and the scope was the following:

- Safe4337Module.sol
- AddModulesLib.sol

We began our review by using static analysis tools, namely Wake. We then took a deep dive into the logic of the contracts. For testing and fuzzing, we have involved Wake testing framework. We implemented tests for account creation when first User Operation is sent. See Appnedix B for more information about the test. During the review, we paid special attention to:

- ensuring the code follows ERC-4337 standard,
- detecting possible malicious behavior between the simulation and execution,
- ensuring access controls are robust enough and compatible with ERC-4337 flow,
- looking for common issues such as data validation.

Our review resulted in 7 findings, ranging from Info to warning severity. The codebase is well designed and follows the best practices and ERC-4337 standard. The complexity of the codebase is hidden in the flow described in ERC-4337 standard. The codebase is well-documented and tested.

Ackee Blockchain recommends Safe:

- update naming convention to achieve full ERC-4337 compatibility,
- address all other reported issues.

See Revision 1.0 for the system overview of the codebase.

Revision 1.1

The client provided the repository with the updated codebase on the given commit: `1981fbc` [2]. The fix review was performed on November 8, 2023. The codebase was updated according to the recommendations from the previous revision.

See the summary of the findings for the current status of issues.

See Revision 1.1 for the review of the updated codebase and additional information we consider essential for the current scope.

Revision 1.2

The client discovered two low issues that caused the contract to be not 100% compatible with current safe functionalities. The client provided the updated repository with the updated codebase on the given commit: `0371f5ac` [3].

The fix review was performed on November 27, 2023.

See the summary of the findings for the current status of issues, including the client's discovered issues.

See Revision 1.2 for the review of the updated codebase and additional information we consider essential for the current scope.

Revision 2.0

The client updated the codebase to version `0.2.0` as a response to the potential issues, where a user's wallet may pay more gas than expected (see M1) if the malicious actor changes one of the User Operation parameters. The updated codebase was delivered on the given commit: `c366d82` [4], and the review was performed on December 4 and 5, 2023.

During the review, we emphasized the new way of handling a User Operation structure and its signing. We also implemented tests where we checked that the variable `operationData` is encoded as expected, and we compared the result with other possible implementations of the same functionality mentioned in the Pull request #177.

The codebase is very well documented and tested. The only problem we discovered is one factically incorrect in-code comment, which was immediately fixed by the client on the commit: `25779b5` [5].

See the summary of the findings for the current status of issues, including the client's discovered issues.

See Revision 2.0 for the review of the updated codebase and additional information we consider essential for the current scope.

[1] full commit hash: 53211cc3dc12a0f5ffadb2bdce9089403fdc8bdd

[2] full commit hash: 1981fbc63e3850d626074d81d22a198afe64ac03

[3] full commit hash: 0371f5ac81da1a5275e5c5643fb95c1c4dda2121

[4] full commit hash: c366d822c42c656df3988624897a5b88fa83b2d7

[5] full commit hash: 25779b5a5077e109a585993a02c4dad2209ab084

4. Summary of Findings

The following table summarizes the findings we identified during our review. Unless overridden for purposes of readability, each finding contains:

- a *Description*,
- an *Exploit scenario*,
- a *Recommendation* and if applicable
- a *Fix*.

There might often be multiple ways to solve or alleviate the issue, with varying requirements regarding the necessary changes to the codebase. In that case, we will try to enumerate them all, clarifying which solves the underlying issue better (albeit possibly only with architectural changes) than others.

	Severity	Reported	Status
W1: Lack of data validation in the constructor	Warning	1.0	Fixed
W2: Usage of `solc` optimizer	Warning	1.0	Acknowledged
I1: Naming convention does not follow ERC-4337 standard	Info	1.0	Fixed
I2: Missing underscore in the internal function	Info	1.0	Fixed
I3: Contract name is not equal to file name	Info	1.0	Fixed

	Severity	Reported	Status
I4: Contract does not allow to specify `validAfter` and `validUntil` parameters	Info	1.0	Acknowledged
I5: Incorrect documentation	Info	1.0	Fixed
L1: Module does not support contract signatures	Low	1.2	Fixed
L2: Incorrect length of return bytes	Low	1.2	Fixed
M1: User wallet can be forced to pay more gas than expected	Medium	2.0	Fixed
I6: Incorrect in-code comment	Info	2.0	Fixed

Table 2. Table of Findings

ackee | blockchain security

5. Report revision 1.0

5.1. System Overview

This section contains an outline of the audited contracts. Note that this is meant for understandability purposes and does not replace project documentation.

Contracts

Contracts we find important for better understanding are described in the following section.

Safe4337Module

The contract is a Safe Fallback handler and Module at the same time. The contract receives calls forwarded from the `fallback` function inside the Safe contract, and it can call the function `executeTransactionFromModule`. These functionalities are necessary for the contract to work correctly with ERC-4337 contracts.

The contract implements functions defined by ERC-4337 standard. The first one is the `validateUserOp`, which is called by EntryPoint contract during the simulation of User Operation. As defined, it performs several validations:

- The address of `msg.sender` == sender of a User Operation.
- Only two functions can be called in User Operation - `executeUserOp` or `executeUserOpWithErrorString`.
- Only trusted EntryPoint can call Safe (that forwards the call to this contract by fallback).

Another two functions: `executeUserOp` and `executeUserOpWithErrorString` perform transaction execution and are callable only by EntryPoint contract.

The rest of the functions are performing signature validation.

AddModulesLib

The library is used when an account makes the first User Operation, and a wallet has to be created with `InitCode`. In this case, modules must be added to the Safe contract to ensure a correct functionality compatible with ERC-4337.

Actors

This part describes actors of the system, their roles, and permissions.

Safe wallet

The Safe Wallet initiates the User Operation. Moreover, it implements ERC-4337 compatible functions for the User Operation simulation and execution.

EntryPoint

The contract is defined by the ERC-4337 standard, which is called by a bundler of User Operations initiated by account wallets.

5.2. Trust Model

EntryPoint is a main trusted component. Setting the address to a malicious one can cause fatal consequences.

The ERC-4337 standard defines the trust model of the account. Functions only work correctly if they are called by EntryPoint contract, forwarded from Safe contract.

W1: Lack of data validation in the constructor

Impact:	Warning	Likelihood:	N/A
Target:	Safe4337Module	Type:	Data validation

Description

The contract Safe4337Module does not perform any data validation inside the constructor. Even though there is no direct threat, the data validation should be performed to avoid unintended behavior if the mistake goes unnoticed, or to avoid additional cost for a new deployment as there is no setter function for a new `entryPoint`.

Recommendation

Perform contract existing checks, or zero address check, to avoid some unintended mistakes during the contract creation.

Fix 1.1

Zero-address check was added to the constructor.

Go back to Findings Summary

W2: Usage of `solc` optimizer

Impact:	Warning	Likelihood:	N/A
Target:	`**/*`	Type:	Compiler configuration

Description

The project uses `solc` optimizer. Enabling `solc` optimizer [may lead to unexpected bugs](). The Solidity compiler was audited in November 2018, and the audit [concluded]() that the optimizer may not be safe.

Recommendation

Until the `solc` optimizer undergoes more stringent security analysis, opt-out using it. This will ensure the protocol is resilient to any existing bugs in the optimizer.

Fix 1.1

The team acknowledged the warning. The reasons are explained in the documentation:

> After careful consideration, we decided to enable the optimizer for the following reasons:
>
> - The most critical functionality, such as signature checks and replay protection, is handled by the Safe and Entrypoint contracts.
> - The entrypoint contract uses the optimizer.

[Go back to Findings Summary]()

I1: Naming convention does not follow ERC-4337 standard

Impact:	Info	Likelihood:	N/A
Target:	Safe4337Module	Type:	Best practices

Description

The function `validateUserOp` does not follow ERC-4337 standard naming convention in two places:

- Input parameter `requiredPrefund` should be named `missingAcountFunds`.
- Return parameter `validationResult` should be named `validationData`.

Recommendation

Change the names of the mentioned variables to follow the standard ERC-4337. It makes the code more easy to understand for users and developers.

Fix 1.1

Parameters names were changed as proposed.

Go back to Findings Summary

I2: Missing underscore in the internal function

Impact:	Info	Likelihood:	N/A
Target:	Safe4337Module	Type:	Best practices

Description

The function `validateSignatures` is internal, but it does not contain an underscore in its name.

Recommendation

Change the names from `validateSignatures` to `_validateSignatures`. It makes the code more easy to read and understand while auditing or debugging.

Fix 1.1

The name of the function was changed as proposed.

Go back to Findings Summary

I3: Contract name is not equal to file name

Impact:	Info	Likelihood:	N/A
Target:	Safe4337Module	Type:	Best practices

Description

The contract's name is `Safe4337Module`; however, the name of the solidity file is `EIP4337Module.sol`. There is no rule to match the names, but it is good practice, making the orientation in the codebase easier.

Recommendation

Change the name of the file to `Safe4337Module.sol`

Fix 1.1

The name of the file was changed to match the contract's name.

Go back to Findings Summary

I4: Contract does not allow to specify `validAfter` and `validUntil` parameters

Impact:	Info	Likelihood:	N/A
Target:	Safe4337Module	Type:	Best practices

Description

The contact `Safe4337Module` automatically returns `validationResult` from a function `validateUserOp`. Based on the standard, it should contain three encoded parameters:

- aggregator address
- validAfter
- validAfter

Two `validX` parameters are used to specify the lifetime of the User Operation. The contract always returns `0` for both parameters. This behavior is NOT wrong, and it follows ERC-4337, and zero value means the User Operation is valid without any time limitation.

Adding the ability to set values `validAfter` `validAfter` will give more flexibility to the wallet.

Recommendation

Consider adding the ability to change these two parameters by the wallet owner.

Fix 1.1

The issue was acknowledged with the following comment:

> We are choosing not to support this feature at the moment but may implement it in a follow-up revision of the module

[Go back to Findings Summary]

ackee | blockchain security

I5: Incorrect documentation

Impact:	Info	Likelihood:	N/A
Target:	`**/*`	Type:	Documentation

Description

The code snippet in the README.md file in a chapter Setup Flow contains mistakes.

```
* Enable Modules **/
bytes memory initExecutor = ADD_MODULES_LIB_ADDRESS;
bytes memory initData =  abi.encodeWithSignature("enableModules", [
4337_MODULE_ADDRESS, ENTRY_POINT_ADDRESS]);

/** Setup Safe **/
// We do not want to use any payment logic therefore, this is all set to 0
bytes memory setupData = abi.encodeWithSignature("setup", owners,
threshold, initExecutor, initData, 4337_MODULE_ADDRESS, address(0), 0,
address(0));

/** Deploy Proxy **/
bytes memory deployData = abi.encodeWithSignature("createProxyWithNonce",
SAFE_SINGLETON_ADDRESS, setupData, salt);

/** Encode for 4337 **/
bytes memory initCode = abi.encodePacked(SAFE_PROXY_FACTORY_ADDRESS,
deployData);
```

The problem is in `abi.encodeWithSignature` format, where the first parameter must be a complete function signature, including argument types. Instead of the line:

```
abi.encodeWithSignature("createProxyWithNonce", SAFE_SINGLETON_ADDRESS,
setupData, salt);
```

It should be:

```
abi.encodeWithSignature("createProxyWithNonce(address,bytes,uint256)",
SAFE_SINGLETON_ADDRESS, setupData, salt);
```

Recommendation

Change the documentation to the correct format, to make the Setup Flow work correctly for users.

Fix 1.1

The code snippet was fixed.

Go back to Findings Summary

ackee blockchain security

6. Report revision 1.1

No significant changes were performed in the contracts, and no new vulnerabilities were found. All the changes are responding to reported issues.

7. Report revision 1.2

The function `_getOperationData` was added so the not-hashed data can be passed to the main Safe contract to verify the signature. No new issues were discovered.

L1: Module does not support contract signatures

Low severity issue

Impact:	Low	Likelihood:	Low
Target:	Safe4337Module	Type:	Compatibility

The Safe team internally discovered the issue.

Description

The contract does not support verification of contract signatures based on **ERC-1271**. The problem is that the contract does not call a Safe core contract with encoded operation bytes but only with a hash of data and signatures.

```
try ISafe(payable(userOp.sender)).checkSignatures(operationHash, "",
userOp.signature)
```

The encoded operation bytes are necessary for **ERC-1271** signature verification implemented in the Safe core contract.

Fix 1.2

The function `_getOperationData` to get not hashed operation data was added, so the contract can now call Safe core contract with `operationData` data, and it is compatible with **ERC-1271** signatures.

```
try ISafe(payable(userOp.sender)).checkSignatures(operationHash,
operationData, userOp.signature)
```

Go back to Findings Summary

L2: Incorrect length of return bytes

Low severity issue

Impact:	Low	Likelihood:	Low
Target:	Safe4337Module	Type:	Contract logic

The Safe team internally discovered the issue.

Description

The function `executeUserOpWithErrorString` executes a user operation and returns an error message if execution is not successful.

```
if (!success) {
    // solhint-disable-next-line no-inline-assembly
    assembly {
        revert(add(returnData, 0x20), returnData)
    }
}
```

The `revert` OP code works in the following way: `revert(a,b)` returns the data from memory, of size `b` starting from slot `a`. The first parameter in the code snippet is correct, but the second one returns memory offset `returnData` instead of the length `mload(returnData)`.

It is not a serious issue because the offset will always be a bigger number than the length of the data. However, it returns an unnecessary long revert string.

Fix 1.2

The code was rewritten in the following way:

```
if (!success) {
    // solhint-disable-next-line no-inline-assembly
    assembly ("memory-safe") {
        revert(add(returnData, 0x20), mload(returnData))
    }
}
```

Go back to Findings Summary

8. Report revision 2.0

The updated codebase contains two main changes:

- In the current codebase **all** the parameters of the User Operation are signed by the user.
- The parameter `signature` contains values defining the time range of signature validity: `validUntil`, `validAfter`.

Additionally, the new structure `EncodedSafeOpStruct` was defined and it is used internally for computing EIP-712 struct-hash.

M1: User wallet can be forced to pay more gas than expected

Medium severity issue

Impact:	Medium	Likelihood:	Medium
Target:	Safe4337Module	Type:	Gas griefing

The Safe team internally discovered the issue.

Description

The contract does not include all the User Operation parameters in its signature mechanism. Two missing parameters are: `initCode` and `paymasterAndData`. Because those parameters are not included in the signed data structure, changing them in the User Operation is possible. If other (signed) parameters are correct, the User Operation will be executed. A malicious actor can cause a user's wallet to pay more gas fees than expected.

Exploit scenario

Bob sends his User Operation to the mempool with a variable `paymasterAndData` pointing to a paymaster contract. Alice front-runs Bob and sends the same User Operation to the mempool but with **empty** `paymasterAndData` variable. If Bob's wallet has any available Ether, the wallet will pay for the transaction execution instead of a paymaster contract.

If `initCode` is changed by Alice, it can perform some on-chain operation before the actual wallet initialization, which will cause Bob's wallet to spend more gas than expected.

Fix 2.0

The logic was rewritten, and all the User Operation parameters are now signed.

Go back to Findings Summary

I6: Incorrect in-code comment

Impact:	Info	Likelihood:	N/A
Target:	Safe4337Module	Type:	Documentation

Description

The internal function `_validateSignatures` contains the following NatSpec in-code documentation:

```
/**
     * @dev Validates that the user operation is correctly signed. Reverts
if signatures are invalid.
     * @param userOp User operation struct.
     * @return validationData An integer indicating the result of the
validation.
*/
```

The first line claims that `revert` will happen when signatures are invalid; however, this is incorrect. Instead of revert, when signatures are invalid, the `validationData` variable will be returned with a first byte `0x01` representing a signature validation fail.

Recommendation

Update the in-code documentation by deleting the sentence or changing it.

Fix 2.0

The comment was changed.

[Go back to Findings Summary]

Appendix A: How to cite

Please cite this document as:

Ackee Blockchain, Safe: ERC-4337 Module, 5.12.2023.

ackee | blockchain security

Appendix B: Wake outputs

B.1. Tests

The following test simulates contract creation with initCode in the first User Operation initiated by an account.

```
class Safe4337Fuzz(FuzzTest):
    safes: Dict[Safe, Tuple[List[Account], int]]
    safe_nonces: DefaultDict[Safe, int]
    erc4337_module: Safe4337Module
    add_modules_lib: AddModulesLib

    def __init__(self) -> None:
        self.entry_point =
IEntryPoint("0x0576a174D229E3cFA37253523E645A78A0C91B57")
        self.safe_proxy_factory =
Account("0x4e1DCf7AD4e460CfD30791CCC4F9c8a4f820ec67")
        self.safe_singleton =
Account("0x41675C099F32341bf84BFc5382aF534df5C7461a")

    def pre_sequence(self) -> None:
        self.safes = {}
        self.safe_nonces = defaultdict(int)
        self.erc4337_module = Safe4337Module.deploy(self.entry_point)
        self.add_modules_lib = AddModulesLib.deploy()

    def _random_safe_op(self) -> bytes:
        return Abi.encode_call(
            Safe4337Module.executeUserOp,
            [random_account(), 0, random_bytes(0, 32), random.choice([0,
1])],
        )

    def _random_user_op(self, safe: Safe, init_code: bytes) ->
UserOperation:
        op = UserOperation(
            safe.address,
            self.safe_nonces[safe],
```

```
            bytearray(init_code),
            bytearray(self._random_safe_op()),
            1_000_000,
            1_000_000,
            0,
            0,
            0,
            bytearray(b""),
            bytearray(b""),
        )
        op.signature = bytearray(self._sign_user_op(safe, op))
        return op

    def _sign_user_op(self, safe: Safe, user_op: UserOperation) -> bytes:
        accounts, threshold = self.safes[safe]
        signers = sorted(random.sample(accounts, threshold))
        hash = self.erc4337_module.getOperationHash(
            safe,
            user_op.callData,
            user_op.nonce,
            user_op.preVerificationGas,
            user_op.verificationGasLimit,
            user_op.callGasLimit,
            user_op.maxFeePerGas,
            user_op.maxPriorityFeePerGas,
            self.entry_point,
        )
        signature = bytearray()
        for signer in signers:
            signature += signer.sign_hash(hash)
        return signature

    def _run_user_op(self, user_op: UserOperation):
        with must_revert(IEntryPoint.ValidationResult) as e:
            self.entry_point.simulateValidation(user_op)

        tx = self.entry_point.handleOps([user_op], Address(1))
        #print(tx.call_trace)
        #breakpoint()

    @flow(max_times=10)
    def flow_create_safe_execute_op(self):
```

```
        owners_count = random.randint(2, 5)
        owners = random.sample(default_chain.accounts, owners_count)
        threshold = random.randint(1, owners_count)

        init_data = Abi.encode_call(
            AddModulesLib.enableModules,
            [[self.erc4337_module, self.add_modules_lib]],
        )
        setup_data = Abi.encode_call(
            Safe.setup,
            [owners, threshold, self.add_modules_lib, init_data,
self.erc4337_module, Address(0), 0, Address(0)],
        )
        deploy_data = Abi.encode_call(
            SafeProxyFactory.createProxyWithNonce,
            [self.safe_singleton, setup_data, 0],
        )
        init_code = Abi.encode_packed(["address", "bytes"],
[self.safe_proxy_factory, deploy_data])

        with must_revert(IEntryPoint.SenderAddressResult) as e:
            self.entry_point.getSenderAddress(init_code)
        safe = Safe(e.value.sender)

        self.safes[safe] = (owners, threshold)

        user_op = self._random_user_op(safe, init_code)
        self._run_user_op(user_op)

        assert len(safe.code) > 0

    @flow()
    def flow_execute_op(self):
        if len(self.safes) == 0:
            return

        safe = random.choice(list(self.safes.keys()))
        user_op = self._random_user_op(safe, b"")
        self._run_user_op(user_op)
```

Thank You

Ackee Blockchain a.s.

Prague, Czech Republic

hello@ackeeblockchain.com

https://twitter.com/AckeeBlockchain