ackee | blockchain security

Brahma

Console v2

by Ackee Blockchain

9.10.2023

ackee | blockchain security

Contents

1. Document Revisions

0.1	Draft report	5.10.2023
1.0	Final report	5.10.2023
1.1	Fix review	9.10.2023

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 Woke 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
Jan Kalivoda	Lead Auditor
Michal Prevratil	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

Brahma Console is a custody and DeFi execution environment built with Safe as custody rails, enabling granular access control with transaction policies and roles, as well as automated execution.

Revision 1.0

Brahma engaged Ackee Blockchain to perform a security review of the Brahma protocol with a total time donation of 8 engineering days in a period between September 25 and October 5, 2023 with Jan Kalivoda as the lead auditor.

The audit was performed on the commit 3578883 [1] and the scope was the following:

- AddressProvider.sol
- AddressProviderService.sol
- Constants.sol
- ExecutorPlugin.sol
- PolicyValidator.sol
- SafeDeployer.sol
- SafeEnabler.sol
- SafeModerator.sol
- SafeModeratorOverridable.sol
- TransactionValidator.sol
- ExecutorRegistry.sol
- PolicyRegistry.sol

- WalletRegistry.sol
- SafeHelper.sol
- TypeHashHelper.sol

We began our review by using static analysis tools, namely Woke. We then took a deep dive into the logic of the contracts. For testing and fuzzing, we involved Woke testing framework. We prepared a fuzz test covering the whole project. This yielded the H1 and M1 issues.

During the review, we paid special attention to:

- checking and testing signature validation of all kinds,
- checking the possibility of manipulating registries,
- checking the possibility of replay attacks,
- ensuring the guards can not lead to DoS or be bypassed,
- ensuring access controls are not too relaxed or too strict,
- detecting possible reentrancies in the code,
- looking for common issues such as data validation.

Our review resulted in 7 findings, ranging from Info to High severity. The most severe one, discovered by the fuzz test, poses a possibility of denial of service (see H1). Otherwise, the codebase is of high quality and well-designed.

Ackee Blockchain recommends Brahma:

- update the documentation according to the new codebase,
- address all other reported issues.

See Revision 1.0 for the system overview of the codebase.

Revision 1.1

The review was done on the given commit: `4589ec4` [2] and the scope was only the findings. All important issues were fixed.

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

[1] full commit hash: `3578883f3d9677118b7ed442ceda83d66b11af38`

[2] full commit hash: `4589ec4732b4b113673b93436e074042a776fe32`

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
H1: Console permanent denial of service	High	1.0	Fixed
M1: `_isGuardBeingRemoved` check dysfunctional	Medium	1.0	Fixed
L1: Console guard can be enabled with zero policy	Low	1.0	Fixed
W1: Authorized addresses can not be deauthorized	Warning	1.0	Acknowledged
W2: `CallType` different order than Safe `Operation`	Warning	1.0	Fixed
W3: Registry addresses can not be changed	Warning	1.0	Acknowledged

	Severity	Reported	Status
I1: Outdated documentation	Info	1.0	Acknowledged

Table 2. Table of Findings

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.

AddressProvider

The `AddressProvider` contract manages and updates addresses that are used by other contracts in the system. Specifically, registries and authorized contracts. The contract is managed by an address that should represent governance. Other contracts interact with the `AddressProvider` with the help of the `AddressProviderService` contract by inheriting it.

Registries

The protocol is using 3 registries:

- `ExecutorRegistry` - The contract manages executors that are allowed to execute transactions for subaccounts.
- `PolicyRegistry` - The contract manages policy commits per account. The policy can be updated if the commit is zero and the `msg.sender` is SafeDeployer. Or if the `msg.sender` is the owner of the account that is going to have an updated policy, or if the `msg.sender` is a wallet and is updating the policy for itself.
- `WalletRegistry` - The contract allows registering wallets and subaccounts.

SafeDeployer

The contract manages the deployment of Safe accounts and their configuration to console or subaccounts accordingly.

SafeModerator

The contract represents a Safe guard that validates transactions (see TransactionValidator) that have to be executed on subaccounts.

SafeModeratorOverridable

The contract is similar to the SafeModerator, but it is intended for console accounts.

TransactionValidator

The contract provides hooks for validation. The subject of validation is for example policy commit (see PolicyValidator) or signatures.

PolicyValidator

The contract checks the validator's signature against the policy commit for the corresponding account and if it is not expired.

ExecutorPlugin

The contract represents a Safe module and additional execution possibility. It checks the executor's signature and validity for the corresponding account.

Actors

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

Safe

Safe architecture is used as business logic for the protocol.

Console

A Safe account that manages subaccounts and can act as a standalone Safe or Console with additional logic.

Subaccount

A safe account with an enabled guard (see SafeModerator). Allows enabling ExecutorPlugin and thus alongside policy commits it provides fine-grained execution control.

Executor

An account that is allowed to use ExecutorPlugin.

Brahma governance

An account that can add other authorized addresses or registries, but not update them if they already exists.

5.2. Trust Model

Users have to trust Brahma when using their frontend/backend, otherwise from a smart contract perspective, the protocol is well-designed and leverages best practices of trustless protocols. However, the governance is in charge of the authorized addresses registry which can affect users positively but also negatively.

H1: Console permanent denial of service

High severity issue

Impact:	High	Likelihood:	Medium
Target:	`**/*`	Type:	Denial of service

Description

This issue is a combination of the M1 and L1 issues. If the `SafeModeratorOverridable` guard is enabled on a Safe console and the current policy is the zero policy, the console account is permanently locked and cannot send any transaction.

Exploit scenario

A Safe console account is created with the zero policy. The owners of the Safe account decide to enforce a new policy. The `SafeModeratorOverridable` guard is enabled on the console. Then, the owners of the console send a transaction, changing the policy. However, the transaction reverts as the validation fails because the current policy is the zero policy (see L1). Furthermore, the guard cannot be disabled because of the issue in the `TransactionValidator._isGuardBeingRemoved` function (see M1).

Recommendation

Fix M1 and consider fixing L1 issues.

Fix 1.1

Fixed by fixing the issues.

Go back to Findings Summary

M1: `_isGuardBeingRemoved` check dysfunctional

Medium severity issue

Impact:	Medium	Likelihood:	Medium
Target:	TransactionValidator	Type:	Logic error

Description

The `SafeModeratorOverridable` guard can be enabled on a Safe account to enforce a custom policy. The guard can be later removed without the active policy verification. For this purpose, the `TransactionValidator` contract contains the `_isGuardBeingRemoved` function to detect a transaction that removes the guard.

```
function _isGuardBeingRemoved(address _to, uint256 _value, bytes memory
_data, Enum.Operation _operation)
    internal
    view
    returns (bool)
{
    if (msg.sender == _to && _value == 0 && _operation ==
Enum.Operation.Call) {
        if (SafeHelper._GUARD_REMOVAL_CALLDATA_HASH == keccak256(_data)) {
            return true;
        }
    }

    return false;
}
```

The first part of the condition checks that the Safe account performs a call to itself to change the guard to the zero address. However, `msg.sender` is never the Safe account because the `TransactionValidator` contract is not directly called by the Safe account but through the `SafeModeratorOverridable`

contract.

```
TransactionValidator(AddressProviderService._getAuthorizedAddress(_TRANSACT
ION_VALIDATOR_HASH))
    .validatePreTransactionOverridable(
    TransactionValidator.SafeTransactionParams({
        from: msg.sender,
        to: to,
        value: value,
        data: data,
        operation: operation,
        safeTxGas: safeTxGas,
        baseGas: baseGas,
        gasPrice: gasPrice,
        gasToken: gasToken,
        refundReceiver: refundReceiver,
        signatures: signatures,
        msgSender: msgSender
    })
);
```

As a consequence, the transaction disabling the guard is never detected in this function and a trusted validator signature is required. The signature is only generated and appended to the transaction if the given transaction complies with the current policy. However, disabling the guard may be against the policy, depending on the implementation of the trusted validator. Furthermore, if the current policy is the zero policy (as described in L1) the guard cannot be removed at all and the Safe account is locked forever without a possibility to send a transaction.

Exploit scenario

The `SafeModeratorOverridable` guard is enabled on a Safe account. Later, the owners of the Safe account decide to remove the guard and send a transaction removing the guard. However, the `_isGuardBeingRemoved` function fails to detect the transaction as a guard removal transaction and the

transaction is rejected if the trusted validator signature is not provided. The signature may or may not be provided depending on the implementation of the trusted validator.

Recommendation

Replace the `msg.sender` expression in the `_isGuardBeingRemoved` function with the value of the `SafeTransactionParams.from` field.

Fix 1.1

The function was renamed from `_isGuardBeingRemoved` to `_isConsoleBeingOverriden` and fixed by adding another parameter `from` to the function that is used instead of `msg.sender`.

Go back to Findings Summary

L1: Console guard can be enabled with zero policy

Low severity issue

Impact:	Low	Likelihood:	Medium
Target:	PolicyValidator, SafeDeployer	Type:	Data validation

Description

The Brahma protocol relies on admin accounts called consoles to manage Safe subaccounts with configured policies. The console accounts may be EOA or Safe multisig accounts. In the latter case, it is also possible to configure and enforce a policy on a console account. However, the `SafeDeployer` contract allows for creation of Safe console accounts with the zero policy (the policy is unset).

```
function deployConsoleAccount(address[] calldata _owners, uint256
_threshold, bytes32 _policyCommit, bytes32 _salt)
    external
    nonReentrant
    returns (address _safe)
{
    _safe = _createSafe(_owners, _setupConsoleAccount(_owners, _threshold),
_salt);

    if (_policyCommit != bytes32(0)) {

PolicyRegistry(AddressProviderService._getRegistry(_POLICY_REGISTRY_HASH)).
updatePolicy(
            _safe, _policyCommit
        );
    }
    emit ConsoleAccountDeployed(_safe);
}
```

The `SafeModeratorOverridable` contract must be enabled as a guard for a

console account to enforce a policy on the console account. This is performed as a Safe multisig transaction without interactions with the Brahma protocol. As a consequence, the `SafeModeratorOverridable` guard may be enabled on a console with zero policy.

Policies are checked using an external trusted validator and the signature of the validator is verified by the `PolicyValidator` contract. However, this contract reverts if the policy is not set (zero policy).

```
bytes32 policyHash =

PolicyRegistry(AddressProviderService._getRegistry(_POLICY_REGISTRY_HASH)).
commitments(account);
if (policyHash == bytes32(0)) {
    revert NoPolicyCommit();
}
```

This effectively causes denial of service for the console account. The only workaround is to disable the `SafeModeratorOverridable` guard, as this operation should not require the trusted validator signature. However, as described in the H1 issue, the detection of a transaction disabling the guard contains a bug, making the denial of service permanent.

Exploit scenario

Owners of a Safe console account want to enforce a new policy. The current policy is the zero policy. The owners enable the `SafeModeratorOverridable` guard. After that, they want to set the new policy, but the transaction reverts because changing a policy requires the signature of the trusted validator. The signature is verified by the `PolicyValidator` contract, but this contract reverts because the current policy is not set.

Recommendation

Consider requiring a non-zero policy when creating a console account. If this is not an option, provide a helper function that enables the `SafeModeratorOverridable` guard and requires a policy to be set.

Fix 1.1

The policy commit is passed as a parameter to the `deployConsoleAccount` function and if it is not zero, the policy is updated in the deployment transaction. Otherwise, the policy is not updated, because the project wants to allow users more flexibility.

> Console allows users to have optional policies on the main safe. Users can choose enable policy validation as a feature to enable additional security or they can choose to maintian complete uninhibited control on their main account.
>
> — Brahma Finance

Go back to Findings Summary

W1: Authorized addresses can not be deauthorized

Impact:	Warning	Likelihood:	N/A
Target:	WalletRegistry.sol	Type:	Logic

Description

The wallets and subaccounts can be registered but not deregistered. This can be potentially an issue in case of some disaster to keep the console secure.

Recommendation

Ensure this is wanted behavior.

Go back to Findings Summary

W2: `CallType` different order than Safe `Operation`

Impact:	Warning	Likelihood:	N/A
Target:	Types	Type:	Code quality

Description

The `CallType` enum in the `Types.sol` file defines call type enum values in a different order than the `Operation` enum from Safe contracts.

```
enum CallType {
    STATICCALL,
    DELEGATECALL,
    CALL
}
```

```
enum Operation {
    Call,
    DelegateCall
}
```

Recommendation

For consistency, consider changing the order of the `CallType` members to follow the `Operation` enum order.

Fix 1.1

The order was correctly changed.

Go back to Findings Summary

W3: Registry addresses can not be changed

Impact:	Warning	Likelihood:	N/A
Target:	AddressProvider.sol	Type:	Disaster recovery

Description

Registry addresses can not be changed in AddressProvider. It's neither good nor bad, but it has certain implications.

Disallowing that strengthens the trust model, since these addresses are immutable and can not be changed maliciously. On the other hand, in case of some disaster, it blocks potential help for the protocol.

Recommendation

Inform users about the implications of the design that leverages the trust model.

Go back to Findings Summary

I1: Outdated documentation

Impact:	Info	Likelihood:	N/A
Target:	`**/*`	Type:	Best practices

Description

The project provides up-to-date inlined and repository documentation, however, the official documentation is outdated.

Recommendation

Update the documentation to match the new version of the protocol.

Go back to Findings Summary

6. Report revision 1.1

The codebase was changed to address reported issues in this report and a new contract was added. It was internally discovered that off-chain signing can potentially bypass policy verification, so the new `ConsoleFallbackHandler` contract now replaces the default `CompatibilityFallbackHandler` contract and has additional policy validation in the `isValidSignature` function for ERC-1271 verification.

No new issues were introduced.

Appendix A: How to cite

Please cite this document as:

Ackee Blockchain, Brahma: Console v2, 9.10.2023.

Appendix B: Glossary of terms

The following terms might be used throughout the document:

Superclass/Ancestor of C

A contract that C inherits/derives from.

Subclass/Child of C

A contract that inherits/derives from C.

Syntactic contract

A Solidity contract. May have an inheritance chain, and may be deployed.

Deployed contract

An EVM account with non-zero code. If its source was written in Solidity, it was created through at least one syntactic contract. If that contract had superclasses (parents), it would be composed of multiple syntactic contracts.

Init/initialization function

A non-constructor function that serves as an initializer. Often used in upgradeable contracts.

External entrypoint

A `public` or `external` function.

Public/Publicly-accessible function/entrypoint

An `external` or `public` function that can be successfully executed by any network account.

Mutating function

A non-`view` and non-`pure` function.

Appendix C: Woke outputs

C.1. Tests

The following part of our test suite shows an example of functions used in fuzzing campaign flows.

```
def get_validator_signature(self, execution_params: ExecutionParams, expiry_epoch: int,
    return self.trusted_validator.sign_structured(
        ValidationParams(
            execution_params,
            policy,
            expiry_epoch,
        ),
        domain={
            "name": "PolicyValidator",
            "version": "1.0",
            "chainId": default_chain.chain_id,
            "verifyingContract": self.policy_validator,
        }
    )

def safe_exec_transaction(self, safe: IGnosisSafe, safe_tx: SafeTx, *, append_validator
    signatures = self.sign_safe_tx(safe, safe_tx)
    sender = random_account()

    # either console with enabled SafeModeratorOverridable or subaccount
    if append_validator_signature and (safe in self.guarded_consoles or safe not in sel
        expiry_epoch = default_chain.blocks["latest"].timestamp + random_int(1, 100)
        validator_signature = self.get_validator_signature(
            safe_tx_to_execution_params(safe_tx, safe, Account(0)),
            expiry_epoch,
            self.policies[safe],
        )
        signatures += validator_signature + len(validator_signature).to_bytes(4) + expi

    tx = safe.execTransaction(*safe_tx, signatures, from_=sender)
    assert tx.return_value
    return tx
```

and the second part of the test suite shows an example of the flows themselves.

```python
@flow()
def flow_subaccount_send_tx(self):
    if sum(len(v) for v in self.subaccounts.values()) == 0:
        return

    console = random.choice([c for c in self.consoles if len(self.subaccounts[c]) > 0])
    subaccount = random.choice(self.subaccounts[console])
    safe_tx = random_safe_tx()

    self.safe_exec_transaction(subaccount, safe_tx)

    logger.info(f"Sent tx {safe_tx} from subaccount {subaccount} of console {console}")

@flow()
def flow_update_policy(self):
    if len(self.consoles) == 0:
        return

    if any(len(v) > 0 for v in self.subaccounts.values()) and random.random() < 0.5:
        # update subaccount policy
        console = random.choice([c for c in self.consoles if len(self.subaccounts[c]) >
        acc = random.choice(self.subaccounts[console])
    else:
        # update console policy
        console = random.choice(self.consoles)
        acc = console

    policy = random_bytes(32, 32, predicate=lambda b: b != b"\x00" * 32)
    self.set_policy(console, acc, policy)

    logger.info(f"Updated policy of {acc} to {policy.hex()}")

@flow()
def flow_add_executor(self):
    consoles = [c for c in self.consoles if len(self.subaccounts[c]) > 0]
    if len(consoles) == 0:
        return

    console = random.choice(consoles)
    subaccount = random.choice(self.subaccounts[console])
    executor = random_account()

    if isinstance(console, IGnosisSafe):
        safe_tx = SafeTx(
            self.executor_registry,
            0,
            Abi.encode_call(
                ExecutorRegistry.registerExecutor,
                [subaccount, executor],
```

ackee | blockchain security

Thank You

Ackee Blockchain a.s.


Prague, Czech Republic


hello@ackeeblockchain.com


https://twitter.com/AckeeBlockchain