sponsor: "Brahma" slug: "2023-10-brahma" date: "2023-11-21" title: "Brahma" findings: "https://github.com/code-423n4/2023-10-brahma-findings/issues" contest: 295

Overview

About C4

Code4rena (C4) is an open organization consisting of security researchers, auditors, developers, and individuals with domain expertise in smart contracts.

A C4 audit is an event in which community participants, referred to as Wardens, review, audit, or analyze smart contract logic in exchange for a bounty provided by sponsoring projects.

During the audit outlined in this document, C4 conducted an analysis of the Brahma smart contract system written in Solidity. The audit took place between October 13 — October 20, 2023.

Wardens

54 Wardens contributed reports to Brahma:

1. bronze_pickaxe
2. Taylor_Webb
3. ABA
4. ladboy233
5. imare
6. T1MOH
7. merlin
8. klau5
9. 0xmystery
10. immeas
11. 0xAnah
12. radev_sw
13. alexzoid
14. GREY-HAWK-REACH (Kose, dimulski, aslanbek and Habib0x)
15. niroh
16. hihen
17. mgf15
18. 0xVolcano
19. SovaSlava
20. rahul
21. hunter_w3b
22. Bube
23. 0xbrett8571
24. emerald7017
25. DadeKuma
26. invitedtea
27. catellatech
28. 0xweb3boy
29. fouzantanveer
30. Arz
31. sorrynotsorry
32. Alra
33. rvierdiiev
34. Raihan
35. 0x11singh99
36. lsaudit
37. m4ttm
38. SAAJ
39. castle_chain
40. 0xSmartContract
41. 0xdice91
42. xiao
43. JCK
44. K42
45. LinKenji
46. albahaca
47. digitizeworx
48. Myd
49. Bauchibred
50. ZanyBonzy
51. 0xDetermination

This audit was judged by 0xsomeone.

Final report assembled by thebrittfactor.

About Brahma Console

Brahma Console is a specialised custody and execution environment for on-chain execution. With Console, users can segregate risk and delegate operations, as well as execute and automate transactions on any dApp, through UI or API.

Summary

The C4 analysis yielded an aggregated total of 4 unique vulnerabilities. Of these vulnerabilities, 0 received a risk rating in the category of HIGH severity and 4 received a risk rating in the category of MEDIUM severity.

Additionally, C4 analysis included 8 reports detailing issues with a risk rating of LOW severity or non-critical. There were also 8 reports recommending gas optimizations.

All of the issues presented here are linked back to their original finding.

Scope

The code under review can be found within the C4 Brahma repository, and is composed of 16 smart contracts written in the Solidity programming language and includes 883 lines of Solidity code.

In addition to the known issues identified by the project team, a Code4rena bot race was conducted at the start of the audit. The winning bot, **IIIIIII-bot** from warden IIIIIII, generated the Automated Findings report and all findings therein were classified as out of scope.

Severity Criteria

C4 assesses the severity of disclosed vulnerabilities based on three primary risk categories: high, medium, and low/non-critical.

High-level considerations for vulnerabilities span the following key areas when conducting assessments:

- Malicious Input Handling
- Escalation of privileges
- Arithmetic
- Gas use

For more information regarding the severity criteria referenced throughout the submission review process, please refer to the documentation provided on the C4 website, specifically our section on Severity Categorization.

Medium Risk Findings (4)

[M-01] A safe that has been created using version `1.40=<` will not be compatible with Brahma

Submitted by bronze_pickaxe, also found by imare

Safe's created outside of the Brahma ecosystem should be able to seamlessly integrate into the `Brahma`. This Safe should call `WalletRegistry.registerWallet` to register. After registration, this safe will be a `consoleAccount` and should be able to use the same functionality that all the other `consoleAccounts` have.

However, Safe's that have been created using version `1.4.0=<` are not fully compatible with Brahma. This is because, in version 1.4.0, `IERC165` support has been added to the `GuardManager.sol`, this is the code added:

```
+ if (guard != address(0)) {
+ require(Guard(guard).supportsInterface(type(Guard).interfaceId), "GS300");
+ }
```

This means that every Safe that has been created using Safe's contract version 1.40 and up, can only add guards that support the `EIP-165` interface, as read from the CHANGELOG.md

Proof of Concept

Consider the following:

- Alice has a safe setup.
- Alice wants to integrate their safe into the Brahma ecosystem.
- Alice calls `WalletRegistry.registerWallet` and this call succeeds.
- Alice decides they want to implement the guard contract provided by the Brahma ecosystem, `SafeModeratorOverridable.sol`
- Alice calls `GnosisSafe.setGuard(address(SafeModeratorOverridable))`.
- This will fail because of this new require statement in Safe contracts `v1.4.0=<`:

```
function setGuard(address guard) external authorized {
        if (guard != address(0)) {
            require(Guard(guard).supportsInterface(type(Guard).interfaceId), "GS300");
        }
```

Because the `SafeModeratorOverridable.sol` does not support the `EIP-165` interface:

```
source: contracts/src/core/SafeModeratorOverridable.sol

contract SafeModeratorOverridable is AddressProviderService, IGuard {
```

This means that every Safe created with version 1.4.0 or up, can not implement the guard contract, which is a fundamental part of the way the `ConsoleAccounts` function.

Recommended Mitigation Steps

Add support for the `EIP-165` interface or update the Safe contracts used in Brahma from 1.3.0 to the most recent version.

Assessed type

Context

0xsomeone (judge) decreased severity to Medium and commented:

> This appears to be a contender for "best" as it clearly pinpoints the flaw (i.e. Gnosis Safe instances created externally rather than via the code) as well as versions (i.e. `>=1.4.0`) the bug is applicable to.

0xad1onchain (Brahma) confirmed

0xad1onchain (Brahma) commented:

> Fixed, added the recommended IERC165 support.

[M-02] `SubAccount` operator can steal funds via the gas refund mechanism

Submitted by Taylor_Webb, also found by ladboy233

The `SubAccount` operator can steal funds from the SubAccount Gnosis Safe via the Gnosis Safe gas refund mechanism, since the trusted validator signatures do not include the following parameters that are included in the `Safe.sol` `execTransaction` function:

- `uint256 safeTxGas`
- `uint256 baseGas`
- `uint256 gasPrice`
- `address gasToken`
- `address payable refundReceiver`

This allows the operator to call `execTransaction` with these parameters set to send all the Ether or a chosen ERC-20 token held in the SubAccount Gnosis Safe to the operator's address.

Proof of Concept

One of the operators, who is an owner on the SubAccount Gnosis Safe, calls `execTransaction` on the Gnosis Safe with valid `to`, `value`, `data`, `operation`, and `signatures` parameters which is allowed by the current policy configured on the SubAccount, and therefore the trusted validator will also sign.

However, the operator will also populate `execTransaction` function parameters related to gas refunds (`safeTxGas, baseGas, gasPrice, gasToken` and `refundReceiver`), to values that transfer to themselves Ether or ERC-20 tokens held in the SubAccount Gnosis Safe.

The transaction is executed, and in the `handlePayment` function call within the Gnosis Safe execution, the Ether or ERC-20 tokens will be transferred to the address the operator specified in the `refundReceiver` parameter.

Recommended Mitigation Steps

Include the `safeTxGas, baseGas, gasPrice, gasToken and refundReceiver` parameters of the transaction in the trusted validated signature that is verified.

Assessed type

Invalid Validation

0xad1onchain (Brahma) confirmed via duplicate Issue #484

0xsomeone (judge) decreased severity to Medium and commented:

> The Warden has demonstrated a potential avenue via which the `SafeModerator` security checks in relation to the interaction performed can be "bypassed". Effectively, operators of the sub-account can extract any funds held by the sub-account by taking advantage of the gas refund mechanism.
>
> Given that the price-per-gas unit can be arbitrarily specified, a minuscule waste of gas can result in a significant amount of funds being extracted. This particular avenue also bypasses the `SafeModerator` security checks.
>
> As such, the Warden has showcased a way to actively affect funds within a sub-account that is not intended behaviour by the Sponsor.

0xad1onchain (Brahma) commented:

> Fixed, added missing params to transaction validation struct.

[M-03] Protocol is not `EIP712` compliant: incorrect typehash for `Validation` and `Transaction` structures

Submitted by ABA, also found by 0xmystery

When implementing `EIP712`, among other things, for data structures that will be a part of signing message, a typehash must be defined.

The structure typehash is defined as: `typeHash = keccak256(encodeType(typeOf(s)))`

Where `encodeType` is the type of a struct that is encoded as: `name ‖ "(" ‖ member₁ ‖ "," ‖ member₂ ‖ "," ‖ … ‖ memberₙ ")"`

And each member is written as: `type ‖ " " ‖ name`.

The project uses 2 structures on the signed data `Transaction` and `Validation` declared in `TypeHashHelper`:

```
/**
 * @notice Structural representation of transaction details
 * @param operation type of operation
 * @param to address to send tx to
 * @param account address of safe
 * @param executor address of executor if executed via executor plugin, address(0) if executed via execTransaction
 * @param value txn value
 * @param nonce txn nonce
 * @param data txn callData
 */
struct Transaction {
    uint8 operation;
    address to;
    address account;
    address executor;
    uint256 value;
    uint256 nonce;
    bytes data;
}


/**
 * @notice Type of validation struct to hash
 * @param expiryEpoch max time till validity of the signature
 * @param transactionStructHash txn digest generated using TypeHashHelper._buildTransactionStructHash()
 * @param policyHash policy commit hash of the safe account
 */
struct Validation {
    uint32 expiryEpoch;
    bytes32 transactionStructHash;
    bytes32 policyHash;
}
```

However, the precalculated typehash for each of the structure are of different structures:

1. For `Transaction` the hash is actually calculated on a now removed `ExecutionParams` structure:

```
/**
 * @notice EIP712 typehash for transaction data
 * @dev keccak256("ExecutionParams(address to,uint256 value,bytes data,uint8 operation,address account,address executor,uint256
 */
bytes32 public constant TRANSACTION_PARAMS_TYPEHASH =
    0xfd4628b53a91b366f1977138e2eda53b93c8f5cc74bda8440f108d7da1e99290;
```

2. For `Validation` the hash is calculated on another old, removed structure `ValidationParams`:

```
    /**
     * @notice EIP712 typehash for validation data
     * @dev keccak256("ValidationParams(ExecutionParams executionParams,bytes32 policyHash,uint32 expiryEpoch)ExecutionParams(addres
     */
    bytes32 public constant VALIDATION_PARAMS_TYPEHASH =
        0x0c7f653e0f641e41fbb4ed1440c7d0b08b8d2a19e1c35cfc98de2d47519e15b1;
```

POC

The issue went undetected in the initial development phase, and can be verified, because the tests still use the old hashes:

```
    function testValidateConstants() public {
        assertEq(
            TypeHashHelper.TRANSACTION_PARAMS_TYPEHASH,
            keccak256(
                "ExecutionParams(address to,uint256 value,bytes data,uint8 operation,address account,address executor,uint256 nonce)
            )
        );
        assertEq(
            TypeHashHelper.VALIDATION_PARAMS_TYPEHASH,
            keccak256(
                "ValidationParams(ExecutionParams executionParams,bytes32 policyHash,uint32 expiryEpoch)ExecutionParams(address to,u
            )
        );
    }
```

The specific test can be verified by running:

```
forge test --fork-url "https://eth-mainnet.g.alchemy.com/v2/<ALCHEMY_API_KEY>" -vvv --ffi --match-test testValidateConstants
```

Impact

Protocol is not EIP712 compliant, which will result in issues with integrators.

Tools Used

An online `keccak256` checker for validating that the those hashes are not actually for the correct structures.

Recommendations

Modify the structure typehash (and tests) to point to the correct structures.

0xad1onchain (Brahma) confirmed

0xsomeone (judge) commented:

> The Warden has demonstrated that the contract is non-compliant with EIP-712 as the type-hashes defined within it are outdated and incorrect.
>
> The Warden has articulated what the potential impact is and, while slightly brief, the recommendation the Warden provided is correct.
>
> As a result, I fully accept this submission as satisfactory as well as the best of its kind. Issue #182 will be awarded partial credit, given that it managed to pinpoint a single flaw in a single type-hash rather than identify that multiple type hashes were incorrect.

0xad1onchain (Brahma) commented:

> Fixed, amended EIP 712 struct.

[M-04] Module transactions will always fail because incompatible with Safe 1.5.0

Submitted by T1MOH, also found by merlin and klau5

Lines of code

https://github.com/code-423n4/2023-10-brahma/blob/main/contracts/src/core/SafeModerator.sol#L80-L86
https://github.com/code-423n4/2023-10-brahma/blob/main/contracts/src/core/SafeModeratorOverridable.sol#L86-L92

Impact

GnosisSafe has concept of module. Trusted modules of GnosisSafe contract can execute arbitrary transactions without signatures once enabled. That's the main purpose of protocol: to allow the execution of some defined action (like swap on DEX) without signatures via registering such an action in Policy commitment. Then, execute this action via Executor or Main Account without need of delegatee's signatures, and also allow one to register SubAccounts and manage them via Main Account.

Therefore, ExecutorPlugin and Main Account are enabled as modules. The main problem is that all transactions executed via module will fail, meaning SubAccount is just another GnosisSafe without additional utility. When assessing the severity, I took into consideration protocol specific: protocol is about adding new functionality.

> Users also have access to SafeSub-accounts that reduce their risk from the protocol by isolating their interactions.

But core functionality of protocol regarding ExecutorPlugin and SubAccounts doesn't work. Due to this reasoning I submit it as High, despite there is no financial loss.

Proof of Concept

Let's take a look on how tx from module is executed:

First, `guard.checkModuleTransaction()` is called, which returns `bytes32 guardHash`, and then transaction is executed:

https://github.com/safe-global/safe-contracts/blob/810fad9a074837e1247ca24ac9e7f77a5dffce19/contracts/base/ModuleManager.sol#L82-L104

```
    function execTransactionFromModule(
        address to,
        uint256 value,
        bytes memory data,
        Enum.Operation operation
    ) public virtual returns (bool success) {
        // Only whitelisted modules are allowed.
        require(msg.sender != SENTINEL_MODULES && modules[msg.sender] != address(0), "GS104");
        // Execute transaction without further confirmations.
        address guard = getGuard();

        bytes32 guardHash;
        if (guard != address(0)) {
@>          guardHash = Guard(guard).checkModuleTransaction(to, value, data, operation, msg.sender);
        }
        success = execute(to, value, data, operation, type(uint256).max);

        if (guard != address(0)) {
            Guard(guard).checkAfterExecution(guardHash, success);
        }
        if (success) emit ExecutionFromModuleSuccess(msg.sender);
        else emit ExecutionFromModuleFailure(msg.sender);
    }
```

In context of Brahma, Guard contract is SafeModerator or SafeModeratorOverridable. They implement the incorrect interface IGuard, so the function `checkModuleTransaction()` returns nothing.

https://github.com/code-423n4/2023-10-brahma/blob/main/contracts/interfaces/external/IGnosisSafe.sol#L110-L116

```
    function checkModuleTransaction(
        address to,
        uint256 value,
        bytes memory data,
        Enum.Operation operation,
        address module
    ) external;
```

Here is PoC in Remix to ensure that function reverts when trying to decode output while the function doesn't return anything:

```
// SPDX-License-Identifier: MIT

pragma solidity ^0.8.0;

interface WithoutReturn {
      function checkModuleTransaction() external;
}

interface WithReturn {
      function checkModuleTransaction() external returns(bytes32);
}

contract Test {

  function test() external returns(bytes32) {

    address calledContract = address(new ContractWithoutReturn());

    // Reverts on decoding
    bytes32 result = WithReturn(calledContract).checkModuleTransaction();
    return result;
  }
}

contract ContractWithoutReturn is WithoutReturn {

  function checkModuleTransaction() external {}

}
```

As a result, the function `GnosisSafe.execTransactionFromModule()` will always revert.

Tools Used

Remix

Recommended Mitigation Steps

Update interface IGuard:

```
    function checkModuleTransaction(
        address to,
        uint256 value,
        bytes memory data,
        Enum.Operation operation,
        address module
-   ) external;
+   ) external returns(bytes32);
```

Note:

The protocol is meant to use 1.5.0; the feature with `callback to guard in module transactions` was introduced in 1.5.0 recently. I also advise to add tests for different safe Versions; now only 1.3.0 is used in tests

The sponsor said they took interface from this PR because GnosisSafe 1.5.0. is still in development. In further discussion, they noted mistake in the interface.

Assessed type

Error

0xad1onchain (Brahma) commented:

> Protocol is clearly marked to use safe `1.3.0` and has been mentioned in codebase natspac, tests and clarified multiple times on discord chat.

0xad1onchain (Brahma) confirmed and commented:

> Valid, thanks for informing us about this issue. Safe `1.5.0` is still a PR and their specifications keep changing. We will remove support for it until it's released.

0xsomeone (judge) decreased severity to Medium and commented:

> The Sponsor initially maintained that Gnosis Safe v1.5.0 is of no concern, however, the documentation of the `SafeModerator::checkModuleTransaction` / `SafeModeratorOverridable::checkModuleTransaction` clearly states that it has been introduced to facilitate support for v1.5.0.
>
> However, as the Sponsor has clearly specified, version 1.5.0 of Gnosis Safe is actively undergoing changes meaning that the `interface` (although highly unlikely) is subject to change; rendering no course of action as *correct* except for simply marking that `1.5.0` is not supported until proper support for it has been introduced.
>
> I think the confusion arises from the fact that the various modules of Brahma (i.e. `SafeModerator`) are meant to be compatible with more versions than the one actually in use by the `SubAccount` systems etc., given that they represent Gnosis Safe modules that can be attached to any instance, further evidenced by the ability to register any wallet in the `WalletRegistry`.
>
> As the main version the system supports is `1.3.0`, which is fully compatible with the present code, I am inclined to judge this exhibit as a "Medium" rather than a "High". To note, the `bytes32` return value was introduced in Gnosis Safe versions `1.4.0` and up, meaning that the exhibit is applicable to a version that has been released and not only to unreleased versions. In general, the bug will manifest when the module is introduced **outside the scope of the repository** to a Gnosis Safe with a version of `1.4.0>=`.
>
> The Sponsor should either:
>
> - Incorporate the return function signature change in their system to support Gnosis Safes `1.4.0>=` and potentially `1.5.0`.
> - Mark versions `1.4.0` and `1.4.1` **in addition to** `1.5.0` as not supported by the system.

0xad1onchain (Brahma) commented:

> Fixed, removed support for safe wallet `v1.5.0`.

Low Risk and Non-Critical Issues

For this audit, 8 reports were submitted by wardens detailing low risk and non-critical issues. The report highlighted below by **immeas** received the top score from the judge.

The following wardens also submitted reports: radev_sw, alexzoid, GREY-HAWK-REACH, Arz, sorrynotsorry, Alra, and rvierdiiev.

Issue Summary

ID	Title
[L-01]	Changing address for `SafeModerator` or `ConsoleFallbackHandler` will break existing `subAccounts`
[L-02]	No opt-out of creating a safe without looping over nonces
[R-01]	It's confusing that two different methods of determining owner are used
[N-01]	Misleading natspec
[N-02]	Wrong bytes in comment
[N-03]	Unused function

[L-01] Changing address for `SafeModerator` or `ConsoleFallbackHandler` will break existing `subAccounts`

During execution of operations on sub-accounts, the fallback handler and guard are not allowed to change:

`TransactionValidator::_checkSubAccountSecurityConfig`:

```
File: contracts/src/core/TransactionValidator.sol

182:        address guard = SafeHelper._getGuard(_subAccount);
183:        address fallbackHandler = SafeHelper._getFallbackHandler(_subAccount);
184:
185:        // Ensure guard has not been disabled
186:        if (guard != AddressProviderService._getAuthorizedAddress(_SAFE_MODERATOR_HASH)) revert InvalidGuard();
187:
188:        // Ensure fallback handler has not been altered
189:        if (fallbackHandler != AddressProviderService._getAuthorizedAddress(_CONSOLE_FALLBACK_HANDLER_HASH)) {
190:            revert InvalidFallbackHandler();
191:        }
```

Here, the guard and fallback handler are verified unchanged to what is defined in the address provider service.

The issue is that if governance changes the implementations of either of these, this check will fail for all existing sub-accounts; since they are created with what was the guard and fallback handler at time of safe creation.

The sub-accounts are blocked until the console account updates the guard or fallback handler on the sub-account.

Impact

Changing the address of either `SafeModerator` or `ConsoleFallbackHandler` will break every existing `subAccount`. It also forces the owning console accounts to upgrade to the new implementations without any opt-in.

Recommendation

Consider storing the values in `pre`-validation and do the comparison against the stored values in `post`-validation. That way, they are independent of any changes from the protocol.

[L-02] No opt-out of creating a safe without looping over nonces

When creating new safes, in either subAccounts or main console accounts, a nonce is used to create the salt used by the SafeDeployer:

`SafeDeployer::_createSafe` and `_genNonce`:

```
228:         uint256 nonce = _genNonce(ownersHash, _salt);
229:         do {
230:             try IGnosisProxyFactory(gnosisProxyFactory).createProxyWithNonce(gnosisSafeSingleton, _initializer, nonce)
231:             returns (address _deployedSafe) {
232:                 _safe = _deployedSafe;
233:             } catch Error(string memory reason) {
234:                 // KEK
235:                 if (keccak256(bytes(reason)) != _SAFE_CREATION_FAILURE_REASON) {
236:                     // A safe is already deployed with the same salt, retry with bumped nonce
237:                     revert SafeProxyCreationFailed();
238:                 }
239:                 emit SafeProxyCreationFailure(gnosisSafeSingleton, nonce, _initializer);
240:                 nonce = _genNonce(ownersHash, _salt);
241:             } catch {
242:                 revert SafeProxyCreationFailed();
243:             }
244:         } while (_safe == address(0));

...

253:     function _genNonce(bytes32 _ownersHash, bytes32 _salt) private returns (uint256) {
254:         return uint256(keccak256(abi.encodePacked(_ownersHash, ownerSafeCount[_ownersHash]++, _salt, VERSION)));
255:     }
```

If the wallet creation fails, it retries with a new nonce (and thus a new salt) until it succeeds. This opens up an unlikely, but possible, attack vector where a rich malevolent actor could front run wallet creation by creating a lot of wallets increasing the gas cost for the victim due to the iterations.

This of course costs even more gas for the attacker as they need to actually create the wallets making this attack unlikely; but it's still possible.

Since the wallets created by the attacker would be identical, a front run here doesn't really do anything else than what the victim wanted; creating a wallet. Hence, the wallet created by a possible attacker would work just as well for the victim.

Recommendation

Consider adding a boolean flag if the user wishes to iterate over nonces when creating a safe. That way, a user can opt-in to having the looping logic or not.

[R-01] It's confusing that two different methods of determining owner are used

`ExecutorRegistry::registerExecutor` and `deRegisterExecutor` uses two different methods of determining if `msg.sender` is the owner of the `subAccount`:

```
File: contracts/src/core/registries/ExecutorRegistry.sol

40:         if (!_walletRegistry.isOwner(msg.sender, _subAccount)) revert NotOwnerWallet();

55:         if (_walletRegistry.subAccountToWallet(_subAccount) != msg.sender) revert NotOwnerWallet();
```

`WalletRegistry::isOwner` simply does the same check, `subAccountToWallet[_subAccount] == _wallet`.

Using two different methods is confusing.

Recommendation

Consider using the same method in both calls.

[N-01] Misleading natspec

`WalletRegistry::registerWallet`:

```
File: contracts/src/core/registries/WalletRegistry.sol

31:    /**
32:     * @notice Registers a wallet
33:     * @dev Can only be called by safe deployer or the wallet itself
34:     */
```

> `Can only be called by safe deployer or the wallet itself`

It cannot be called by safe deployer, as it only instructs the wallet to call it upon setup (as it only registers `msg.sender`, it would then be the safe deployer address registered as a wallet).

[N-02] Wrong bytes in comment

While declaring `SafeHelper._FALLBACK_REMOVAL_CALLDATA_HASH`:

```
File: contracts/src/libraries/SafeHelper.sol

44:     * abi.encodeCall(IGnosisSafe.setFallbackHandler, (address(0))) = 0xf08a0323000000000000000000000000000000000000000000000000000000000000
```

This is not correct, `abi.encodeCall(IGnosisSafe.setFallbackHandler, (address(0)))` is in fact:

```
0xf08a0323000000000000000000000000000000000000000000000000000000000000
```

The hash is, however, calculated on the correct bytes.

[N-03] Unused function

`AddressProviderService::_onlyGov` is not used anywhere in the code.

Consider removing it as having unused functions can cause confusion and is generally considered bad practice.

0xsomeone (judge) commented:

> Decent report with well-elaborated findings. L-01 may be a design decision (i.e. a critical update is needed due to a vulnerability and thus `SubAccount` implementations are forced to update).

0xad1onchain (Brahma) acknowledged and commented:

> Thanks for the report. I agree with all Non-critical findings.
>
> L-01: SafeModerator and ConsoleFallbackHandler can only be updated by governance and may be done in extreme scenarios. Only registries are kept immutable because they hold state.
>
> L-02: While I totally understand your point that attacker deployed the same bytecode at the deterministic `CREATE2` address, for some reason safe changed their safe deployer implementation and removed the initializer as a part of address determination, leading us to question if there is a possible exploit. We chose to keep it safe and make sure we use an address that we deploy.

Gas Optimizations

For this audit, 8 reports were submitted by wardens detailing gas optimizations. The report highlighted below by **0xAnah** received the top score from the judge.

The following wardens also submitted reports: hihen, mgf15, 0xVolcano, SovaSlava, Raihan, 0x11singh99, and Isaudit.

Introduction

Highlighted below are optimizations exclusively targeting state-mutating functions and view/pure functions invoked by state-mutating functions. In the discussion that follows, only runtime gas is emphasized, given its inevitable dominance over deployment gas costs throughout the protocol's lifetime.

Please be aware that some code snippets may be shortened to conserve space, and certain code snippets may include @audit tags in comments to facilitate issue explanations.

[G-01] Redundant state variable getters

Getters for public state variables are automatically generated by the solidity compiler so there is no need to code them manually as this increases deployment cost.

There are 3 instances of this issue:

1. Make `authorizedAddresses` mapping variable `private` or `internal` since a getter function was defined for it.

   https://github.com/code-423n4/2023-10-brahma/blob/main/contracts/src/core/AddressProvider.sol#L112

```
file: contracts/src/core/AddressProvider.sol

35:    mapping(bytes32 => address) public authorizedAddresses;  //@audit generates a redundant getter function
.
.
.
112:    function getAuthorizedAddress(bytes32 _key) external view returns (address) {
113:        return authorizedAddresses[_key];
114:    }
```

The solidity compiler would automatically create a getter function for the `authorizedAddresses` mapping above since it is declared as a `public` variable. However, a getter function `getAuthorizedAddress()` was also declared in the contract for the same variable; thereby, making two getter functions for the same variable in the contract. We could rectify this issue by making the `authorizedAddresses` variable private or internal (if the variable is to be inherited). The diff below shows how the code could be refactored:

```
diff --git a/contracts/src/core/AddressProvider.sol b/contracts/src/core/AddressProvider.sol
index 5ec51f9..7d3a9d7 100644
--- a/contracts/src/core/AddressProvider.sol
+++ b/contracts/src/core/AddressProvider.sol
@@ -32,7 +32,7 @@ contract AddressProvider {
      * @notice keccak256 hash of authorizedAddress keys mapped to their addresses
      * @dev authorizedAddresses are updatable by governance
      */
-    mapping(bytes32 => address) public authorizedAddresses;
+    mapping(bytes32 => address) internal authorizedAddresses;
```

2. Make the `registries` mapping variable `private` or `internal` since a getter function was defined for it.

   https://github.com/code-423n4/2023-10-brahma/blob/main/contracts/src/core/AddressProvider.sol#L121

```
file: contracts/src/core/AddressProvider.sol

41:     mapping(bytes32 => address) public registries;  //@audit generates a redundant getter function
.
.
.
121:    function getRegistry(bytes32 _key) external view returns (address) {
122:        return registries[_key];
123:    }
```

The solidity compiler would automatically create a getter function for the `registries` mapping above since it is declared as a `public` variable. However, a getter function `getRegistry()` was also declared in the contract for the same variable; thereby, making two getter functions for the same variable in the contract. We could rectify this issue by making the `registries` variable private or internal (if the variable is to be inherited). The diff below shows how the code could be refactored:

```
diff --git a/contracts/src/core/AddressProvider.sol b/contracts/src/core/AddressProvider.sol
index 5ec51f9..0a2381b 100644
--- a/contracts/src/core/AddressProvider.sol
+++ b/contracts/src/core/AddressProvider.sol
@@ -38,7 +38,7 @@ contract AddressProvider {
      * @notice keccak256 hash of registry keys mapped to their addresses
      * @dev registries are only set once by governance and immutable
      */
-    mapping(bytes32 => address) public registries;
+    mapping(bytes32 => address) internal registries;
```

3. Make `registries` mapping variable `private` or `internal` since a getter function was defined for it.

   https://github.com/code-423n4/2023-10-brahma/blob/main/contracts/src/core/registries/WalletRegistry.sol#L63

```
file: contracts/src/core/registries/WalletRegistry.sol

25:    mapping(address wallet => address[] subAccountList) public walletToSubAccountList;  //@audit generates a redundant getter fun

63:    function getSubAccountsForWallet(address _wallet) external view returns (address[] memory) {
64:        return walletToSubAccountList[_wallet];
65;    }
```

The solidity compiler would automatically create a getter function for the `walletToSubAccountList` mapping above since it is declared as a `public` variable. However, a getter function `getSubAccountsForWallet()` was also declared in the contract for the same variable; thereby, making two getter functions for the same variable in the contract. We could rectify this issue by making the `walletToSubAccountList` variable private or internal (if the variable is to be inherited). The diff below shows how the code could be refactored:

```
diff --git a/contracts/src/core/registries/WalletRegistry.sol b/contracts/src/core/registries/WalletRegistry.sol
index afc2b7d..4554b7c 100644
--- a/contracts/src/core/registries/WalletRegistry.sol
+++ b/contracts/src/core/registries/WalletRegistry.sol
@@ -22,7 +22,7 @@ contract WalletRegistry is AddressProviderService {
     /// @notice subAccount addresses mapped to owner wallet
     mapping(address subAccount => address wallet) public subAccountToWallet;
     /// @notice wallet addresses mapped to list of subAccounts
-    mapping(address wallet => address[] subAccountList) public walletToSubAccountList;
+    mapping(address wallet => address[] subAccountList) internal walletToSubAccountList;
     /// @notice address of wallet mapped to boolean indicating if it's a wallet
     mapping(address => bool) public isWallet;
```

[G-02] Refactor external/internal function to avoid unnecessary SLOAD

The function below read storage slots that are previously read in the function that invokes it. We can refactor the external/internal functions to pass cached storage variables as stack variables and avoid the extra storage reads that would otherwise take place in the internal functions.

There is 1 instance for this issue.

The `_updatePolicy()` internal function read storage slots that were read in the `updatePolicy` function that invokes it.

https://github.com/code-423n4/2023-10-brahma/blob/main/contracts/src/core/registries/PolicyRegistry.sol#L42-#L66

```
file: contracts/src/core/registries/PolicyRegistry.sol

35:    function updatePolicy(address account, bytes32 policyCommit) external {
36:        if (policyCommit == bytes32(0)) {
37:            revert PolicyCommitInvalid();
38:        }
39:
40:        WalletRegistry walletRegistry = WalletRegistry(AddressProviderService._getRegistry(_WALLET_REGISTRY_HASH));
41:
42:        bytes32 currentCommit = commitments[account];  //@audit 1st SLOAD
43:
.
.
.
58:        _updatePolicy(account, policyCommit);
59:    }
.
.
.
66:    function _updatePolicy(address account, bytes32 policyCommit) internal {
67:        emit UpdatedPolicyCommit(account, policyCommit, commitments[account]);  //@audit 2nd SLOAD
68:        commitments[account] = policyCommit;
69:    }
```

The `_updatePolicy()` function above reads storage for the value of `commitments[account]`, thereby, incurring an `SLOAD` and also has to re-evaluate the mapping to get the value. However, this storage value `commitments[account]` was also read in the `updatePolicy` function, which invokes `_updatePolicy()`. We could refactor the code such that we replace the storage read and re-evaluation of the mapping `commitments[account]` in the `_updatePolicy` with a cheaper stack read. The diff below shows how the code could be refactored:

```
diff --git a/contracts/src/core/registries/PolicyRegistry.sol b/contracts/src/core/registries/Policy
index 559d5ca..b897fe6 100644
--- a/contracts/src/core/registries/PolicyRegistry.sol
+++ b/contracts/src/core/registries/PolicyRegistry.sol
@@ -55,7 +55,7 @@ contract PolicyRegistry is AddressProviderService {
             revert UnauthorizedPolicyUpdate();
         }
         // solhint-enable no-empty-blocks
-        _updatePolicy(account, policyCommit);
+        _updatePolicy(account, policyCommit, currentCommit);
     }

     /**
@@ -63,8 +63,8 @@ contract PolicyRegistry is AddressProviderService {
      * @param account address of account to set policy commit for
      * @param policyCommit policy commit hash to set
      */
-    function _updatePolicy(address account, bytes32 policyCommit) internal {
-        emit UpdatedPolicyCommit(account, policyCommit, commitments[account]);
+    function _updatePolicy(address account, bytes32 policyCommit, bytes32 currentCommit) internal {
+        emit UpdatedPolicyCommit(account, policyCommit, currentCommit);
         commitments[account] = policyCommit;
     }
 }
```

[G-03] Add unchecked blocks for subtractions where the operands cannot underflow

The solidity compiler introduces some checks to avoid an underflow, but in some scenarios where it is impossible for underflow to occur we can use unchecked blocks to have some gas savings.

There is 1 instance of this issue.

https://github.com/code-423n4/2023-10-brahma/blob/main/contracts/src/core/PolicyValidator.sol#L162-#L166

```
file: contracts/src/core/PolicyValidator.sol

156:    function _decompileSignatures(bytes calldata _signatures)
157:        internal
158:        pure
159:        returns (uint32 expiryEpoch, bytes memory validatorSignature)
160:    {
161:        uint256 length = _signatures.length;
162:        if (length < 8) revert InvalidSignatures();
163:
164:        uint32 sigLength = uint32(bytes4(_signatures[length - 8:length - 4]));  //@audit calculations can be unchecked
165:        expiryEpoch = uint32(bytes4(_signatures[length - 4:length]));  //@audit calculations can be unchecked
166:        validatorSignature = _signatures[length - 8 - sigLength:length - 8];  //@audit calculations can be unchecked
167:    }
```

In the `_decompileSignatures()` function above, the following calculations were made `uint32 sigLength = uint32(bytes4(_signatures[length - 8:length - 4]))`, `expiryEpoch = uint32(bytes4(_signatures[length - 4:length]))` , `validatorSignature = _signatures[length - 8 - sigLength:length - 8]` . However, prior to these calculations, the check `if (length < 8) revert InvalidSignatures()` was made which ensures that the value of the `length` variable should be greater or equal to 8. The code could be refactored as shown in the diff below:

```
diff --git a/contracts/src/core/PolicyValidator.sol b/contracts/src/core/PolicyValidator.sol
index 70d672f..04eb07a 100644
--- a/contracts/src/core/PolicyValidator.sol
+++ b/contracts/src/core/PolicyValidator.sol
@@ -161,9 +161,13 @@ contract PolicyValidator is AddressProviderService, EIP712 {
         uint256 length = _signatures.length;
         if (length < 8) revert InvalidSignatures();

-        uint32 sigLength = uint32(bytes4(_signatures[length - 8:length - 4]));
-        expiryEpoch = uint32(bytes4(_signatures[length - 4:length]));
-        validatorSignature = _signatures[length - 8 - sigLength:length - 8];
+        unchecked {
+         uint32 sigLength = uint32(bytes4(_signatures[length - 8:length - 4]));
+         expiryEpoch = uint32(bytes4(_signatures[length - 4:length]));
+         validatorSignature = _signatures[length - 8 - sigLength:length - 8];
+        }
+
+
     }
```

[G-04] Declaring unnecessary variables

Some variables were defined even though they are used once. Not defining variables can reduce gas cost and contract size.

There is 1 instance of this issue.

https://github.com/code-423n4/2023-10-brahma/blob/main/contracts/src/core/PolicyValidator.sol#L63

```
file: contracts/src/core/PolicyValidator.sol

54:    function isPolicySignatureValid(
55:        address account,
56:        address to,
57:        uint256 value,
58:        bytes memory data,
59:        Enum.Operation operation,
60:        bytes calldata signatures
61:    ) external view returns (bool) {
62:        // Get nonce from safe
63:        uint256 nonce = IGnosisSafe(account).nonce();  @audit unnecessary variable declaration
64:
65:        // Build transaction struct hash
66:        bytes32 transactionStructHash = TypeHashHelper._buildTransactionStructHash(
67:            TypeHashHelper.Transaction({
68:                to: to,
69:                value: value,
70:                data: data,
71:                operation: uint8(operation),
72:                account: account,
73:                executor: address(0),
74:                nonce: nonce
75:            })
76:        );
77:
78:        // Validate signature
79:        return isPolicySignatureValid(account, transactionStructHash, signatures);
80:    }
```

In the `isPolicySignatureValid()` function above, declaring the `nonce` variable is unnecessary, as it was only used once in the function. We could use `IGnosisSafe(account).nonce()` directly instead, and save `3` gas units. The diff below shows how it could be refactored:

```
diff --git a/contracts/src/core/PolicyValidator.sol b/contracts/src/core/PolicyValidator.sol
index 70d672f..0107a64 100644
--- a/contracts/src/core/PolicyValidator.sol
+++ b/contracts/src/core/PolicyValidator.sol
@@ -60,7 +60,6 @@ contract PolicyValidator is AddressProviderService, EIP712 {
         bytes calldata signatures
     ) external view returns (bool) {
         // Get nonce from safe
-        uint256 nonce = IGnosisSafe(account).nonce();

         // Build transaction struct hash
         bytes32 transactionStructHash = TypeHashHelper._buildTransactionStructHash(
@@ -71,7 +70,7 @@ contract PolicyValidator is AddressProviderService, EIP712 {
                 operation: uint8(operation),
                 account: account,
                 executor: address(0),
-                nonce: nonce
+                nonce: IGnosisSafe(account).nonce()
             })
         );
```

[G-05] Calculations should be memoized rather than re-calculating them

In computing, memoization, or memoisation, is an optimization technique used primarily to speed up computer programs by storing the results of expensive function calls to pure functions and returning the cached result when the same inputs occur again.

There is 1 instance of this issue.

Memoize `keccak256(_data)` computation:

https://github.com/code-423n4/2023-10-brahma/blob/main/contracts/src/core/TransactionValidator.sol#L166

```
file: contracts/src/core/TransactionValidator.sol

149:    function _isConsoleBeingOverriden(
150:        address _from,
151:        address _to,
152:        uint256 _value,
153:        bytes memory _data,
154:        Enum.Operation _operation
155:    ) internal pure returns (bool) {
.
.
.
165:        if (_from == _to && _value == 0 && _operation == Enum.Operation.Call) {
166:            if (SafeHelper._GUARD_REMOVAL_CALLDATA_HASH == keccak256(_data)) {  //@audit memoize keccak256(_data) calculation
167:                return true;
168:            } else if (SafeHelper._FALLBACK_REMOVAL_CALLDATA_HASH == keccak256(_data)) {
169:                return true;
170:            }
171:        }
172:
173:        return false;
174:    }
```

In the `_isConsoleBeingOverriden()` function above, the computation `keccak256(_data)` in some scenarios would be computed twice. We could save gas used in the subsequent computation if we memoize the computation. We could cache the result of the calculation the first time in a variable and use the variable in place of the subsequent calculations, so that in scenarios where the `keccak256(_data)` computation would be calculated, we would have a reduced gas cost for the computation to just a stack read. The diff below shows how the code could be refactored:

```
diff --git a/contracts/src/core/TransactionValidator.sol b/contracts/src/core/TransactionValidator.sol
index f31fe06..f02cc78 100644
--- a/contracts/src/core/TransactionValidator.sol
+++ b/contracts/src/core/TransactionValidator.sol
@@ -163,9 +163,10 @@ contract TransactionValidator is AddressProviderService {
          * In case these conditions are met, the guard is being removed, return true
          */
         if (_from == _to && _value == 0 && _operation == Enum.Operation.Call) {
-            if (SafeHelper._GUARD_REMOVAL_CALLDATA_HASH == keccak256(_data)) {
+            bytes32 datahash = keccak256(_data);
+            if (SafeHelper._GUARD_REMOVAL_CALLDATA_HASH == datahash) {
                 return true;
-            } else if (SafeHelper._FALLBACK_REMOVAL_CALLDATA_HASH == keccak256(_data)) {
+            } else if (SafeHelper._FALLBACK_REMOVAL_CALLDATA_HASH == datahash) {
                 return true;
             }
         }
```

[G-06] Multiple accesses of a array should use a local variable cache

The instances below point to the second+ access of a value inside an array, within a function. Caching an array's struct avoids re-calculating the array offsets into memory

Proof of concept

```
struct Person {
    string name;
    uint age;
    uint id;
}

contract NoCacheArrayElement {

    Person[] students;

    function createStudents() external  {
        Person[] memory arrayOfPersons = new Person[](3);
        Person memory newPerson1 = Person("Emmanuel", 15,1);
        Person memory newPerson2 = Person("Faustina", 16,2);
        Person memory newPerson3 = Person("Emmanuela", 14,3);

        arrayOfPersons[0] = newPerson1;
        arrayOfPersons[1] = newPerson2;
        arrayOfPersons[2] = newPerson3;

        _addNewSet(arrayOfPersons);
    }

    function _addNewSet(Person[] memory _persons) internal {
        uint len = _persons.length;
        unchecked {
            for(uint i; i < len; ++i) {
                Person memory newStudent = Person(_persons[i].name, _persons[i].age, _persons[i].id);
                students.push(newStudent);
            }
        }

    }
}
```

```
test for test/NoCacheArrayElement.t.sol:NoCacheArrayElementTest
[PASS] test_createStudents() (gas: 230357)
```

```
struct Person {
    string name;
    uint age;
    uint id;
}

contract CacheArrayElement {

    Person[] students;

    function createStudents() external  {
        Person[] memory arrayOfPersons = new Person[](3);
        Person memory newPerson1 = Person("Emmanuel", 15,1);
        Person memory newPerson2 = Person("Faustina", 16,2);
        Person memory newPerson3 = Person("Emmanuela", 14,3);

        arrayOfPersons[0] = newPerson1;
        arrayOfPersons[1] = newPerson2;
        arrayOfPersons[2] = newPerson3;

        _addNewSet(arrayOfPersons);
    }

    function _addNewSet(Person[] memory _persons) internal {
        uint len = _persons.length;
        unchecked {
            for(uint i; i < len; ++i) {
                Person memory myPerson = _persons[i];
                Person memory newStudent = Person(myPerson.name, myPerson.age, myPerson.id);
                students.push(newStudent);
            }
        }

    }
}
```

```
 test for test/Counter.t.sol:CacheArrayElementTest
[PASS] test_createStudents() (gas: 230096)
```

There is 1 instance of this issue.

https://github.com/code-423n4/2023-10-brahma/blob/main/contracts/src/libraries/SafeHelper.sol#L111

file: contracts/src/libraries/SafeHelper.sol

```
103:    function _packMultisendTxns(Types.Executable[] memory _txns) internal pure returns (bytes memory packedTxns) {
104:        uint256 len = _txns.length;
105:        if (len == 0) revert InvalidMultiSendInput();
106:
107:        uint256 i = 0;
108:        do {
109:            // Enum.Operation.Call is 0
110:            uint8 call = uint8(Enum.Operation.Call);
111:            if (_txns[i].callType == Types.CallType.DELEGATECALL) {
112:                call = uint8(Enum.Operation.DelegateCall);
113:            } else if (_txns[i].callType == Types.CallType.STATICCALL) {
114:                revert InvalidMultiSendCall(i);
115:            }
116:
117:            uint256 calldataLength = _txns[i].data.length;
118:
119:            bytes memory encodedTxn = abi.encodePacked(
120:                bytes1(call), bytes20(_txns[i].target), bytes32(_txns[i].value), bytes32(calldataLength), _txns[i].data
121:            );
122:
123:            if (i != 0) {
124:                // If not first transaction, append to packedTxns
125:                packedTxns = abi.encodePacked(packedTxns, encodedTxn);
126:            } else {
127:                // If first transaction, set packedTxns to encodedTxn
128:                packedTxns = encodedTxn;
129:            }
130:
131:            unchecked {
132:                ++i;
133:            }
134:        } while (i < len);
135:    }
```

The code could be refactored as shown in the diff below:

```
diff --git a/contracts/src/libraries/SafeHelper.sol b/contracts/src/libraries/SafeHelper.sol
index 7830a80..93faa93 100644
--- a/contracts/src/libraries/SafeHelper.sol
+++ b/contracts/src/libraries/SafeHelper.sol
@@ -108,16 +108,17 @@ library SafeHelper {
         do {
             // Enum.Operation.Call is 0
             uint8 call = uint8(Enum.Operation.Call);
-            if (_txns[i].callType == Types.CallType.DELEGATECALL) {
+            Types.Executable memory txns = _txns[i]
+            if (txns.callType == Types.CallType.DELEGATECALL) {
                 call = uint8(Enum.Operation.DelegateCall);
-            } else if (_txns[i].callType == Types.CallType.STATICCALL) {
+            } else if (txns.callType == Types.CallType.STATICCALL) {
                 revert InvalidMultiSendCall(i);
             }

-            uint256 calldataLength = _txns[i].data.length;
+            uint256 calldataLength = txns.data.length;

             bytes memory encodedTxn = abi.encodePacked(
-                bytes1(call), bytes20(_txns[i].target), bytes32(_txns[i].value), bytes32(calldataLength), _txns[i].data
+                bytes1(call), bytes20(txns.target), bytes32(txns.value), bytes32(calldataLength), txns.data
             );

             if (i != 0) {
```

[G-07] Bytes constants are more efficient than string constants

If data can fit into 32 bytes, then you should use bytes32 datatype rather than bytes or strings, as it is cheaper in solidity.

There are 2 instances of this issue.

1. Change `_NAME` and `_VERSION` to `bytes32` data-type in `PolicyValidator` contract.

   https://github.com/code-423n4/2023-10-brahma/blob/main/contracts/src/core/PolicyValidator.sol#L26

   https://github.com/code-423n4/2023-10-brahma/blob/main/contracts/src/core/PolicyValidator.sol#L28

```
file: contracts/src/core/PolicyValidator.sol

26:    string private constant _NAME = "PolicyValidator";
27:
28:    string private constant _VERSION = "1.0";
.
.
.
174:    function _domainNameAndVersion() internal pure override returns (string memory name, string memory version) {
175:        return (_NAME, _VERSION);
175:    }
```

The code could be refactored as shown in the diff below:

```
diff --git a/contracts/src/core/PolicyValidator.sol b/contracts/src/core/PolicyValidator.sol
index 70d672f..a911d39 100644
--- a/contracts/src/core/PolicyValidator.sol
+++ b/contracts/src/core/PolicyValidator.sol
@@ -23,9 +23,9 @@ contract PolicyValidator is AddressProviderService, EIP712 {
     error InvalidSignatures();

     /// @notice EIP712 domain name
-    string private constant _NAME = "PolicyValidator";
+    bytes32 private constant _NAME = "PolicyValidator";
     /// @notice EIP712 domain version
-    string private constant _VERSION = "1.0";
+    bytes32 private constant _VERSION = "1.0";

     constructor(address _addressProvider) AddressProviderService(_addressProvider) {}

@@ -171,7 +171,7 @@ contract PolicyValidator is AddressProviderService, EIP712 {
      * @return name domainName
      * @return version domainVersion
      */
-    function _domainNameAndVersion() internal pure override returns (string memory name, string memory version) {
+    function _domainNameAndVersion() internal pure override returns (bytes32, bytes32) {
         return (_NAME, _VERSION);
     }
 }
```

2. Change `_NAME` and `_VERSION` to `bytes32` data-type in `ExecutorPlugin` contract.

   https://github.com/code-423n4/2023-10-brahma/blob/main/contracts/src/core/ExecutorPlugin.sol#L53

   https://github.com/code-423n4/2023-10-brahma/blob/main/contracts/src/core/ExecutorPlugin.sol#L55

```
file: contracts/src/core/ExecutorPlugin.sol

53:    string private constant _NAME = "ExecutorPlugin";

55:    string private constant _VERSION = "1.0";
.
.
.
159:    function _domainNameAndVersion() internal pure override returns (string memory name, string memory version) {
160:        return (_NAME, _VERSION);
161:    }
```

The code could be refactored as shown in the diff below:

```
diff --git a/contracts/src/core/ExecutorPlugin.sol b/contracts/src/core/ExecutorPlugin.sol
index a790269..f985c61 100644
--- a/contracts/src/core/ExecutorPlugin.sol
+++ b/contracts/src/core/ExecutorPlugin.sol
@@ -50,9 +50,9 @@ contract ExecutorPlugin is AddressProviderService, ReentrancyGuard, EIP712 {
     }

     /// @notice EIP712 domain name
-    string private constant _NAME = "ExecutorPlugin";
+    bytes32 private constant _NAME = "ExecutorPlugin";
     /// @notice EIP712 domain version
-    string private constant _VERSION = "1.0";
+    bytes32 private constant _VERSION = "1.0";

     /// @notice mapping of account to nonce of executors
     mapping(address account => mapping(address executor => uint256 nonce)) public executorNonce;
@@ -156,7 +156,7 @@ contract ExecutorPlugin is AddressProviderService, ReentrancyGuard, EIP712 {
      * @return name domainName
      * @return version domainVersion
      */
-    function _domainNameAndVersion() internal pure override returns (string memory name, string memory version) {
+    function _domainNameAndVersion() internal pure override returns (bytes32, bytes32) {
         return (_NAME, _VERSION);
     }
 }
```

[G-08] Use named returns for local variables of pure functions where it is possible

Proof of Concept

```
library NoNamedReturnArithmetic {

    function sum(uint256 num1, uint256 num2) internal pure returns(uint256){
        return num1 + num2;
    }
}

contract NoNamedReturn {
    using NoNamedReturnArithmetic for uint256;

    uint256 public stateVar;

    function add2State(uint256 num) public {
        stateVar = stateVar.sum(num);
    }
}
```

```
test for test/NoNamedReturn.t.sol:NamedReturnTest
[PASS] test_Increment() (gas: 27639)
```

```
library NamedReturnArithmetic {

    function sum(uint256 num1, uint256 num2) internal pure returns(uint256 theSum){
        theSum = num1 + num2;
    }
}

contract NamedReturn {
    using NamedReturnArithmetic for uint256;

    uint256 public stateVar;

    function add2State(uint256 num) public {
        stateVar = stateVar.sum(num);
    }
}
```

```
test for test/NamedReturn.t.sol:NamedReturnTest
[PASS] test_Increment() (gas: 27613)
```

There are 3 instances of this issue.

1. Use named returns in the `__buildTransactionStructHash()` function.

   https://github.com/code-423n4/2023-10-brahma/blob/main/contracts/src/libraries/TypeHashHelper.sol#L64

```
file: contracts/src/libraries/TypeHashHelper.sol

64:    function _buildTransactionStructHash(Transaction memory txn) internal pure returns (bytes32) {
65:        return keccak256(
66:            abi.encode(
67:                TRANSACTION_PARAMS_TYPEHASH,
68:                txn.to,
69:                txn.value,
70:                keccak256(txn.data),
71:                txn.operation,
72:                txn.account,
73:                txn.executor,
74:                txn.nonce
75:            )
76:        );
77:    }
```

The code could be refactored as shown in the diff below:

```
diff --git a/contracts/src/libraries/TypeHashHelper.sol b/contracts/src/libraries/TypeHashHelper.sol
index df1a5aa..64e0ed0 100644
--- a/contracts/src/libraries/TypeHashHelper.sol
+++ b/contracts/src/libraries/TypeHashHelper.sol
@@ -61,8 +61,8 @@ library TypeHashHelper {
      * @param txn transaction params struct
      * @return transactionStructHash
      */
-    function _buildTransactionStructHash(Transaction memory txn) internal pure returns (bytes32) {
-        return keccak256(
+    function _buildTransactionStructHash(Transaction memory txn) internal pure returns (bytes32 trxHash) {
+        trxHash = keccak256(
             abi.encode(
                 TRANSACTION_PARAMS_TYPEHASH,
                 txn.to,
```

2. Use named returns in the `_buildValidationStructHash()` function.

   https://github.com/code-423n4/2023-10-brahma/blob/main/contracts/src/libraries/TypeHashHelper.sol#L84

```
file: contracts/src/libraries/TypeHashHelper.sol

84:    function _buildValidationStructHash(Validation memory validation) internal pure returns (bytes32) {
85:        return keccak256(
86:            abi.encode(
87:                VALIDATION_PARAMS_TYPEHASH,
88:                validation.transactionStructHash,
89:                validation.policyHash,
90:                validation.expiryEpoch
91:            )
92:        );
93:    }
```

```
diff --git a/contracts/src/libraries/TypeHashHelper.sol b/contracts/src/libraries/TypeHashHelper.sol
index df1a5aa..6df88fc 100644
--- a/contracts/src/libraries/TypeHashHelper.sol
+++ b/contracts/src/libraries/TypeHashHelper.sol
@@ -81,8 +81,8 @@ library TypeHashHelper {
      * @param validation validation params struct
      * @return validationStructHash
      */
-    function _buildValidationStructHash(Validation memory validation) internal pure returns (bytes32) {
-        return keccak256(
+    function _buildValidationStructHash(Validation memory validation) internal pure returns (bytes32 validationHash) {
+        validationHash = keccak256(
             abi.encode(
                 VALIDATION_PARAMS_TYPEHASH,
                 validation.transactionStructHash,
```

3. Use named returns in the `_generateSingleThresholdSignature()` function.

   https://github.com/code-423n4/2023-10-brahma/blob/main/contracts/src/libraries/SafeHelper.sol#L84

```
file: contracts/src/libraries/SafeHelper.sol

87:    function _generateSingleThresholdSignature(address owner) internal pure returns (bytes memory) {
88:        bytes memory signatures = abi.encodePacked(
89:            bytes12(0), // Padding for signature verifier address
90:            bytes20(owner), // Signature Verifier
91:            bytes32(0), // Position of extra data bytes (last set of data)
92:            bytes1(hex"01") // Signature Type - 1 (presigned transaction)
93:        );
94:        return signatures;
95:    }
```

The code could be refactored as shown in the diff below:

```
diff --git a/contracts/src/libraries/SafeHelper.sol b/contracts/src/libraries/SafeHelper.sol
index 7830a80..4497e00 100644
--- a/contracts/src/libraries/SafeHelper.sol
+++ b/contracts/src/libraries/SafeHelper.sol
@@ -84,14 +84,13 @@ library SafeHelper {
      * @param owner Owner of the safe
      * @return signatures bytes array containing single pre validated owner signature
      */
-    function _generateSingleThresholdSignature(address owner) internal pure returns (bytes memory) {
-        bytes memory signatures = abi.encodePacked(
+    function _generateSingleThresholdSignature(address owner) internal pure returns (bytes memory signatures) {
+        signatures = abi.encodePacked(
             bytes12(0), // Padding for signature verifier address
             bytes20(owner), // Signature Verifier
             bytes32(0), // Position of extra data bytes (last set of data)
             bytes1(hex"01") // Signature Type - 1 (presigned transaction)
         );
-        return signatures;
     }

     /**
```

[G-09] Use modifiers rather than invoking functions to perform checks

By using modifiers in place of functions to perform checks, we could reduce the gas cost by up to `40` units. In using modifiers, the solidity compiler would inline the operations of the modifier in the called function. Using functions to perform checks would incur two `JUMPI` instructions, plus the stack setup, which could cost up to `40` gas units. Implementing this change would increase deployment cost, but would reduce the gas cost of the called functions. In the long run, using modifiers would be cheaper. The functions below can also be made more gas efficient by making them `payable` functions:

There are 2 instances of this issue.

1. Convert `_onlyDelegateCall()` function into a modifier.

   https://github.com/code-423n4/2023-10-brahma/blob/main/contracts/src/core/SafeEnabler.sol#L81-#L83

```
file: contracts/src/core/SafeEnabler.sol

43:    function enableModule(address module) public {
44:        _onlyDelegateCall();     @audit use a modifier instead
45:
.
.
.
56:    }
.
.
.
66:    function setGuard(address guard) public {
67:        _onlyDelegateCall();     @audit use a modifier instead
68:
.
.
.
75:    }
.
.
.
81:    function _onlyDelegateCall() private view {    @audit convert to a modifier
82:        if (address(this) == _self) revert OnlyDelegateCall();
83:    }
```

The code could be refactored as shown in the diff below:

```
diff --git a/contracts/src/core/SafeEnabler.sol b/contracts/src/core/SafeEnabler.sol
index 02137cc..8f4b6de 100644
--- a/contracts/src/core/SafeEnabler.sol
+++ b/contracts/src/core/SafeEnabler.sol
@@ -29,6 +29,14 @@ contract SafeEnabler is GnosisSafeStorage {
     /// @dev keccak256("guard_manager.guard.address")
     bytes32 internal constant _GUARD_STORAGE_SLOT = 0x4a204f620c8c5ccdca3fd54d003badd85ba500436a431f0cbda4f558c93c34c8;

+    /**
+     * @notice Validates if the current call being made is DELEGATECALL
+     * @dev reverts if not DELEGATECALL
+     */
+    modifier onlyDelegateCall {
+        if (address(this) == _self) revert OnlyDelegateCall();
+        _;
+    }
     constructor() {
         _self = address(this);
     }
@@ -40,8 +48,7 @@ contract SafeEnabler is GnosisSafeStorage {
      *  Refer https://github.com/safe-global/safe-contracts/blob/186a21a74b327f17fc41217a927dea7064f74604/contracts/base/ModuleMana
      * @param module Module to be whitelisted.
      */
-    function enableModule(address module) public {
-        _onlyDelegateCall();
+    function enableModule(address module) public payable onlyDelegateCall {

         // Module address cannot be null or sentinel.
         // solhint-disable-next-line custom-errors
@@ -63,8 +70,7 @@ contract SafeEnabler is GnosisSafeStorage {
      * @param guard address of the guard
      * @dev delegatecalled during initialization to bypass authorization modifier and set guard on safe
      */
-    function setGuard(address guard) public {
-        _onlyDelegateCall();
+    function setGuard(address guard) public payable onlyDelegateCall {

         bytes32 slot = _GUARD_STORAGE_SLOT;
         // solhint-disable-next-line no-inline-assembly
@@ -74,11 +80,4 @@ contract SafeEnabler is GnosisSafeStorage {
         emit ChangedGuard(guard);
     }
-    /**
-     * @notice Validates if the current call being made is DELEGATECALL
-     * @dev reverts if not DELEGATECALL
-     */
-    function _onlyDelegateCall() private view {
-        if (address(this) == _self) revert OnlyDelegateCall();
-    }
 }
```

2. Convert `_onlyGov()` function into a modifier.

   https://github.com/code-423n4/2023-10-brahma/blob/main/contracts/src/core/AddressProvider.sol#L139-#L141

file: contracts/src/core/AddressProvider.sol

```
52:     function setGovernance(address _newGovernance) external {
53:         _notNull(_newGovernance);
54:         _onlyGov();     @audit use a modifier instead
55:         emit GovernanceTransferRequested(governance, _newGovernance);
56:         pendingGovernance = _newGovernance;
57:     }
.
.
.
77:     function setAuthorizedAddress(bytes32 _key, address _authorizedAddress, bool _overrideCheck) external {
78:         _onlyGov();     @audit use a modifier instead
79:         _notNull(_authorizedAddress);
80:
81:         /// @dev skips checks for supported `addressProvider()` if `_overrideCheck` is true
82:         if (!_overrideCheck) {
83:             /// @dev skips checks for supported `addressProvider()` if `_authorizedAddress` is an EOA
84:             if (_authorizedAddress.code.length != 0) _ensureAddressProvider(_authorizedAddress);
85:         }
86:
87:         authorizedAddresses[_key] = _authorizedAddress;
88:
89:         emit AuthorizedAddressInitialised(_authorizedAddress, _key);
90:     }
.
.
.
97:     function setRegistry(bytes32 _key, address _registry) external {
98:         _onlyGov();     @audit use a modifier instead
99:         _ensureAddressProvider(_registry);
100:
101:        if (registries[_key] != address(0)) revert RegistryAlreadyExists();
102:        registries[_key] = _registry;
103:
104:        emit RegistryInitialised(_registry, _key);
105:    }
.
.
.
139:    function _onlyGov() internal view {    @audit convert to a modifier
140:        if (msg.sender != governance) revert NotGovernance(msg.sender);
141:    }
```

The code could be refactored as shown in the diff below:

```
diff --git a/contracts/src/core/AddressProvider.sol b/contracts/src/core/AddressProvider.sol
index 5ec51f9..6327ab1 100644
--- a/contracts/src/core/AddressProvider.sol
+++ b/contracts/src/core/AddressProvider.sol
@@ -40,6 +40,14 @@ contract AddressProvider {
      */
     mapping(bytes32 => address) public registries;

+    /**
+     * @notice Checks if msg.sender is governance
+     */
+    modifier onlyGov {
+        if (msg.sender != governance) revert NotGovernance(msg.sender);
+        _;
+    }
+
     constructor(address _governance) {
         _notNull(_governance);
         governance = _governance;
@@ -49,9 +57,8 @@ contract AddressProvider {
      * @notice Governance setter
      * @param _newGovernance address of new governance
      */
-    function setGovernance(address _newGovernance) external {
+    function setGovernance(address _newGovernance) external payable onlyGov {
         _notNull(_newGovernance);
-        _onlyGov();
         emit GovernanceTransferRequested(governance, _newGovernance);
         pendingGovernance = _newGovernance;
     }
@@ -74,8 +81,7 @@ contract AddressProvider {
      * @param _authorizedAddress address to set
      * @param _overrideCheck overrides check for supported address provider
      */
-    function setAuthorizedAddress(bytes32 _key, address _authorizedAddress, bool _overrideCheck) external {
-        _onlyGov();
+    function setAuthorizedAddress(bytes32 _key, address _authorizedAddress, bool _overrideCheck) external payable onlyGov {
         _notNull(_authorizedAddress);

         /// @dev skips checks for supported `addressProvider()` if `_overrideCheck` is true
@@ -94,8 +100,7 @@ contract AddressProvider {
      * @param _key key of registry address
      * @param _registry address to set
```

Conclusion

As you embark on incorporating the recommended optimizations, we want to emphasize the utmost importance of proceeding with vigilance and dedicating thorough efforts to comprehensive testing. It is of paramount significance to ensure that the proposed alterations do not inadvertently introduce fresh vulnerabilities, while also successfully achieving the anticipated enhancements in performance.

We strongly advise conducting a meticulous and exhaustive evaluation of the modifications made to the codebase. This rigorous scrutiny and exhaustive assessment will play a pivotal role in affirming both the security and efficacy of the refactored code. Your careful attention to detail, coupled with the implementation of a robust testing framework, will provide the necessary assurance that the refined code aligns with your security objectives and effectively fulfill the intended performance optimizations.

0xsomeone (judge) commented:

> All findings are valid and the report layout is great.

Audit Analysis

For this audit, 26 analysis reports were submitted by wardens. An analysis report examines the codebase as a whole, providing observations and advice on such topics as architecture, mechanism, or approach. The report highlighted below by **niroh** received the top score from the judge.

The following wardens also submitted reports: rahul, hunter_w3b, Bube, 0xbrett8571, emerald7017, DadeKuma, invitedtea, catellatech, 0xweb3boy, fouzantanveer, m4ttm, SAAJ, castle_chain, 0xSmartContract, 0xdice91, xiao, JCK, K42, LinKenji, albahaca, digitizeworx, Myd, Bauchibred, ZanyBonzy, and 0xDetermination.

Overview

Brahma is a smart contract framework enabling digital asset managers to use crypto funds in various DeFi activities, while maintaining granular access control and operation policies to achieve a balance between ease of operation and risk mitigation. This is enabled through the use of multi-sig gnosis safes with enhanced functionality that is integrated through gnosis safe modules and guards.

Brahma achieves its goal mainly through four design constructs:

1. A hierarchical structure comprising Console Accounts and Sub Accounts, where the owners of each Console Account maintain full control of its Sub Accounts.
2. A modular Policy framework through which users can define specific limitations on account activity per account, validated by an external trusted validator.
3. An Executor framework enabling fund managers to grant actors (Executors) limited execution permissions to run selected DeFi activities, while imposing a strict policy on the actions these actors can perform.
4. Additional modules that simplify and automate common DeFi activities such as trading, staking, cost averaging etc. (out of scope for this audit).

Brahma's solution is unique in the landscape of DeFi fund management solutions in that asset managers maintain full custody of their funds throughout the usage of the system. This is achieved through the extensive use of Gnosis Safes as the underlying multisig wallet framework.

Scope

src/libraries/TypeHashHelper.sol - Helper contract that creates an EIP712 compatible struct hash for a transaction struct (used to sign transaction data) and validation struct (used for trusted validator signature)

src/libraries/SafeHelper.sol - Helper contract for interaction with Gnosis Safe functions, such as packing multiple transactions for the Multisend interface.

src/core/TransactionValidator.sol - Called by Brahma Guards and ExecutionPlugin to perform validations before and after running transactions on a Brahma Gnosis Safe.

src/core/SafeModeratorOverridable.sol - A Gnosis Safe Guard (Hook that performs custom validations) applied to Console Accounts that can be disabled or enabled by safe owners.

src/core/SafeModerator.sol - Similar to SafeModeratorOverridable but used with Sub Accounts and does not enable turning on or off (Sub Accounts must have the guard enabled at all times).

src/core/SafeEnabler.sol - A workaround contract to enable setting a module to a gnosis safe while it's being created (typically not possible as only the safe contract can set a module to itself). This is worked around by delegating a call to this contract which adds the module directly to the safe state using the same structure as the original function.

src/core/Constants.sol - Constants used throughout the system. Mainly hashes that are used for mapping the different core contracts on the Address Provider.

src/core/ConsoleFallbackHandler.sol - Used for Console safes that have a policy enabled - replaces the out-of-the-box Gnosis Safe compatibility handler, adding policy validation enforcement.

src/core/AddressProvider.sol - A global mechanism that controls the core contracts of the system. Maps core contracts to fixed hashes. Enable a single governance address to set or update system core contracts. Governance can be handed over to a different address by the current governance (pending acceptance by the new governance).

src/core/PolicyValidator.sol - Used by Brahma Guards to validate policy where one is applied. Checks that the policy signature structure is valid and signed by the authorized validator, that the correct policy hash was signed and that the signature hasn't expired.

src/core/registries/PolicyRegistry.sol - A registry mapping policy hashes to accounts. Verifies that only authorized addresses can set policies (either the SafeDeployer zeroing a policy, a Console Account setting its own policy or a Console Account setting a policy for a Sub Account).

src/core/registries/ExecutorRegistry.sol - A registry of the authorized Executors of each Sub Account. Enables Owner Console Accounts to add/remove Executors to a Sub Account.

src/core/registries/WalletRegistry.sol - Registry for wallets (Console Accounts) and their list of Sub Accounts. Anyone can register an address as a wallet as long as its not already registered. Only the SafeDeployer can register a Sub Account.

src/core/AddressProviderService.sol - An interface enabling read only access to the core contracts registered by the AddressProvider.

src/core/SafeDeployer.sol - Helper contract enabling deployment of Console Accounts and Sub Accounts while handling all configurations required by Brahma (i.e. Enabling Guards, enforcing policies if provided, enabling Console Accounts control of Sub Accounts through the use of Safe Modules etc.)

src/core/ExecutorPlugin.sol - A Gnosis Safe Module that can be applied to Sub Accounts to enable granting execution permissions to registered Executors of a Sub Account.

Approach Taken in evaluating the codebase

In analyzing the codebase we took the following steps:

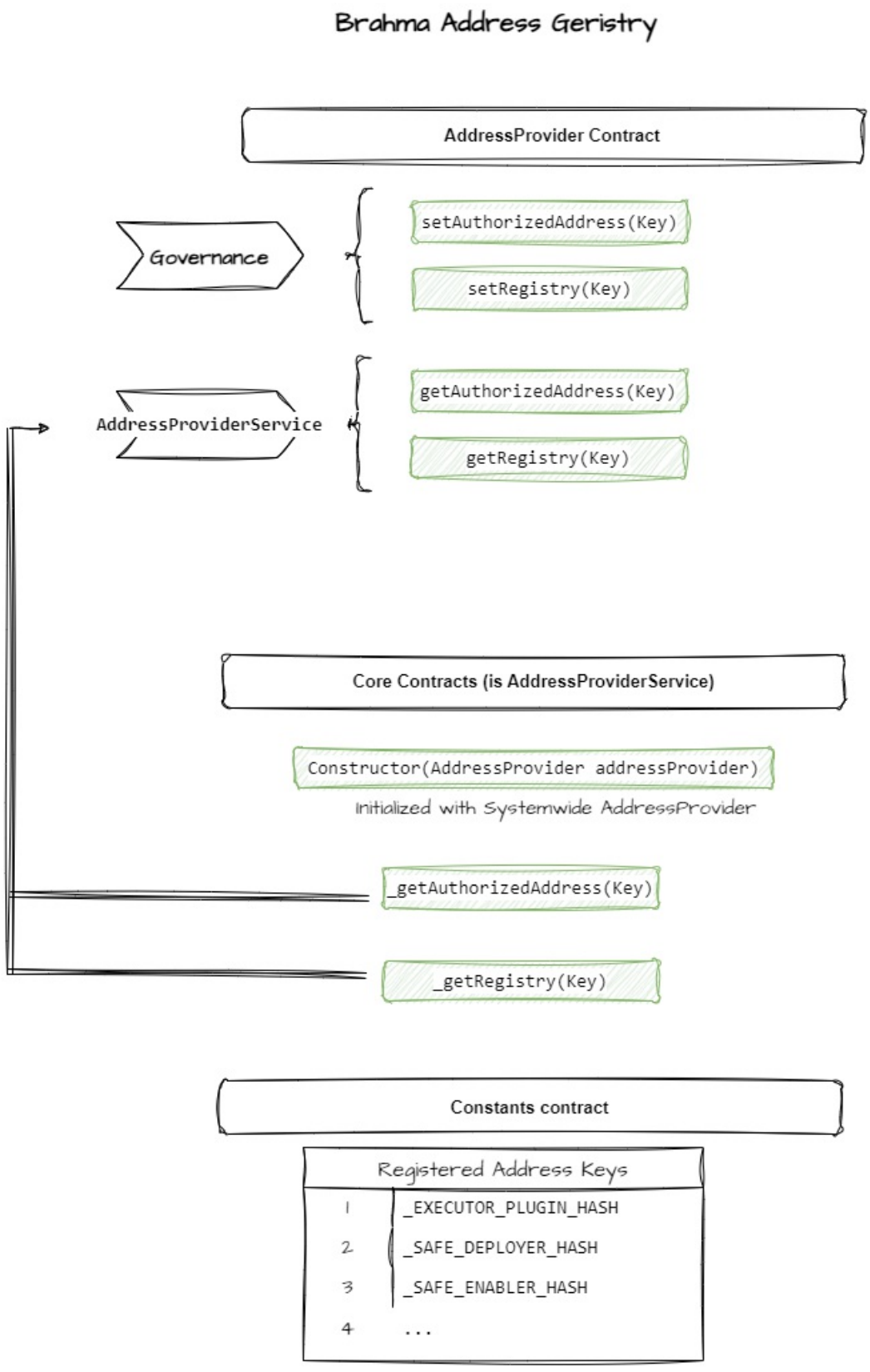
1. **Architecture Review** - Reading available documentation to understand the architecture and main functionality of Brahma Fi.
2. **Compiling code and running provided tests**
3. **Detailed Code Review** - A thorough ode review of the contracts in scope, as well as relevant parts of third party contracts (mainly Gnosis Safe contracts).
4. **Security Analysis** - Defining the main attack surfaces of the codebase (i.e. Safe funds exfiltration, unauthorized transaction executions, etc.).
5. **Additional Testing** - Adding an array of additional tests derived from the potential attack surfaces outlined in the security analysis conducted in the previous step.

Mechanism Review

Brahma's architecture includes several mechanisms working in unison:

1. Core Contract Addresses Registry

This mechanism enables a single governance address to set and update the core contracts used throughout Brahma framework. This registry is managed by the AddressProvider contract. Any system component that needs access to these contracts implements the AddressProviderService interface and is initialized with the (single) AddressProvider contract controlling the system.



2. End User Registries

These registries keep track of Console and Sub Account wallets and their associated relations, security policies and permitted Executors. Three registries exist in the system:

1. **Wallet Registry** - Keeps track of Console (main) Accounts, Sub Accounts and their relations. Any address can register as a wallet (Console Account). Sub

Accounts can be registered only through the SafeDeployer contract.

2. **Policy Registry** - Keeps track of Policy Hashed applied to accounts. Policies can be set by Console Accounts to themselves or to their Sub Accounts.
3. **Executor Registry** - For Sub Accounts that use the ExecutorPlugin, keeps track of the list of Executor addresses approved to run transactions on each Sub Account.

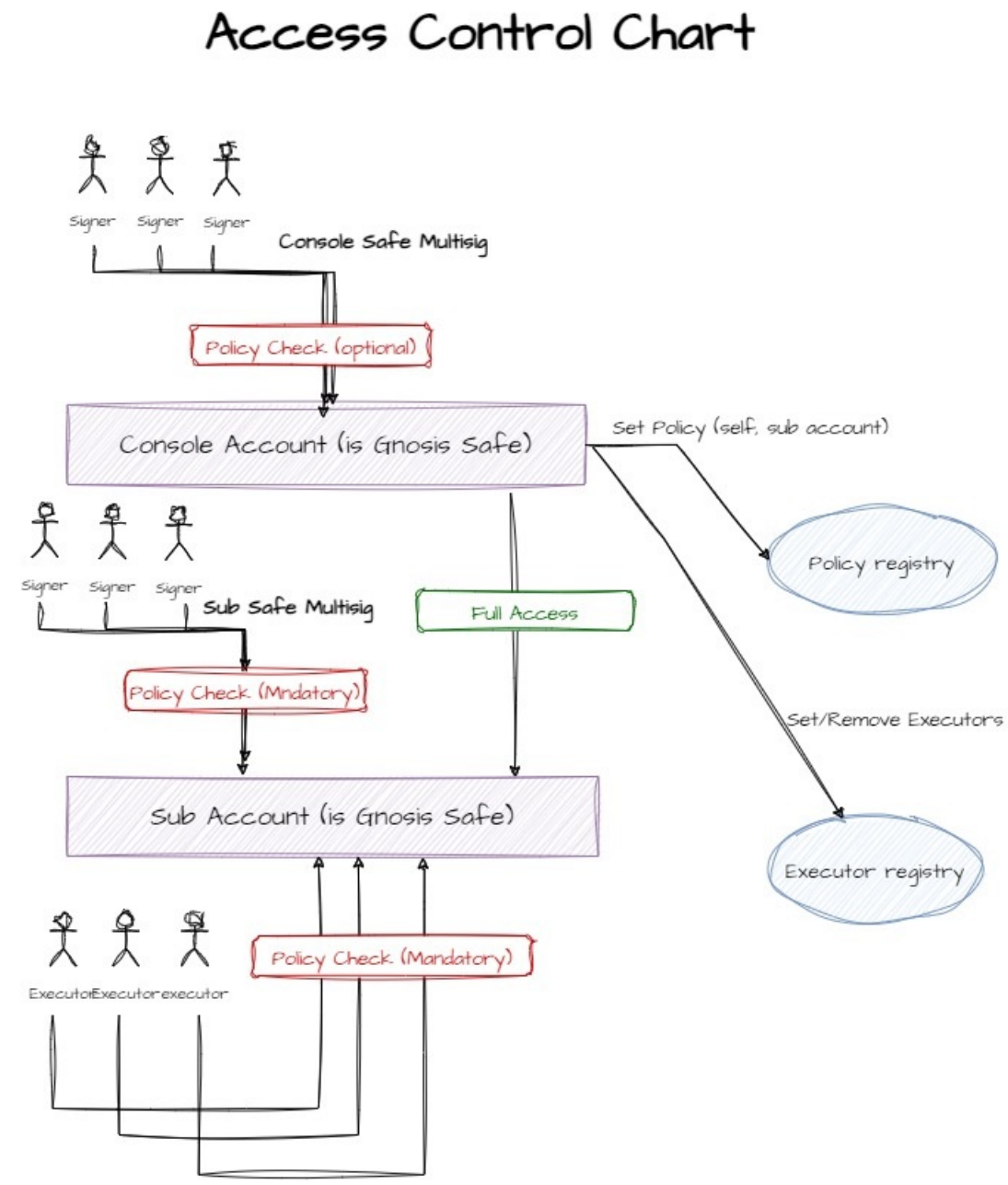
3. Access control

Access control forms the heart of the Brahma protocol. On Brahma there are four mechanisms controlling access that may be combined in different ways:

1. The built in Gnosis Safe multisignatures/threshold system controlling Safes.
2. A policy system through which custom transaction execution policies can be applied per account. Sub Accounts must have a policy assigned to them, while Console Account owners can choose whether or not to enforce a policy. Policies are validated by en external entity (Trusted Validator) whos address is set through the AddressProvider. Transactions send to Accounts with a policy assigned to them must be signed by the trusted validator, along with a hash of the specific policy applied.
3. An Executor Plugin enabling the Console Account controlling a Sub Account to set specific addresses as approved Executors. These addresses can then send transactions to be executed by the Sub Account. These transactions must be signed by both the Executor and the Trusted Validator to be executed.
4. A Console Account can always run transactions on Sub Accounts it controls, bypassing any policies applied to them (but still requiring the Console Account multisig).

Brahma's ACL system is implemented with the help of two Gnosis Safe features: Guards (hooks that enable attaching custom validations to Safes) and Modules (addresses that when registered on a Safe as Modules are allowed to bypass the regular multisignature path and execute transactions on the Safe directly).

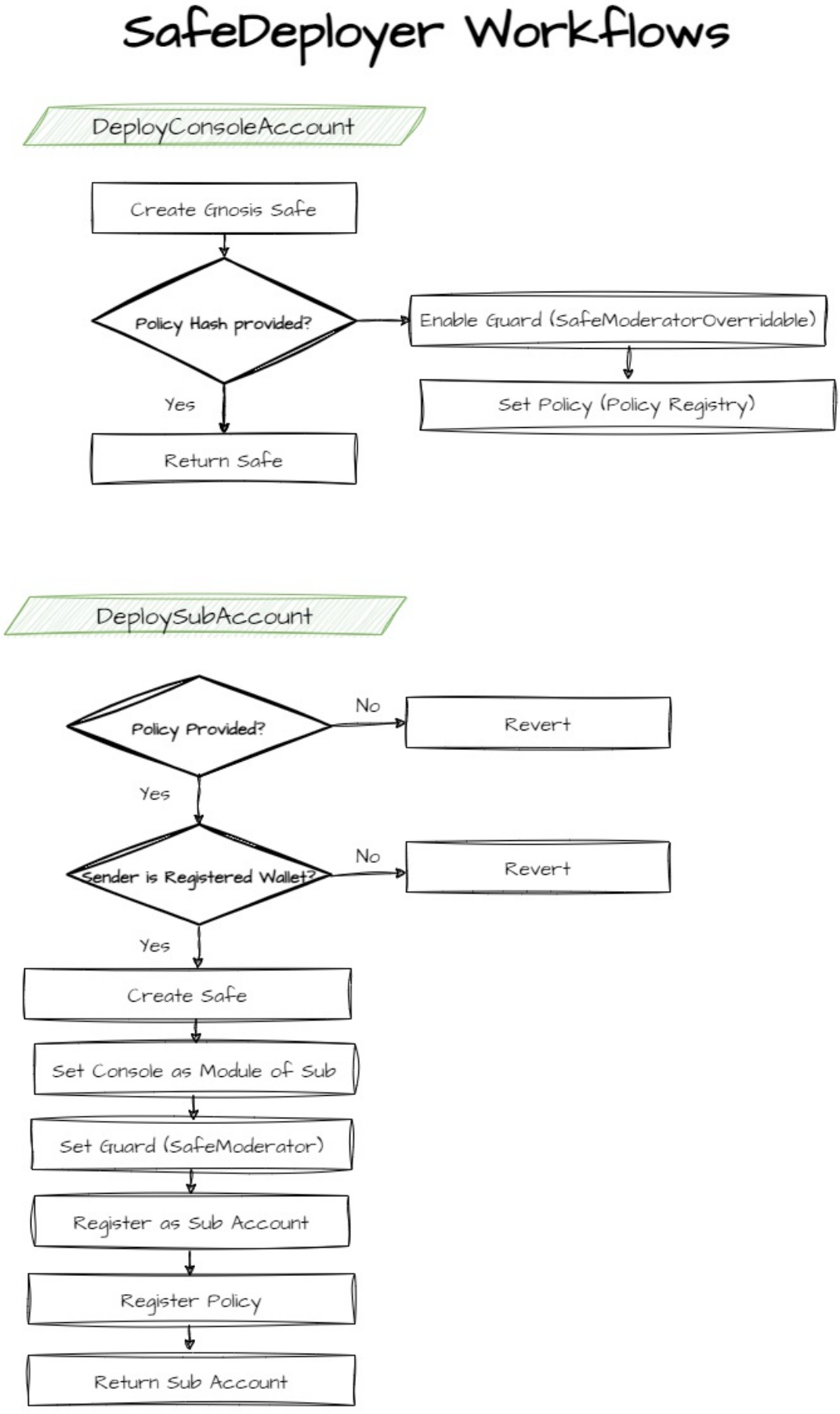
The following diagram illustrates how access control works for Brahma Safes.



4. Helper Contracts

A set of contracts implementing common functionalities to simplify working with the system. These include:

- **SafeDeployer** - Implements the deployment of Console Accounts and Sub Accounts while taking care of all necessary configurations and setups.
- **SafeHelper** - Helper functions for working with Safes (such as running a transaction on a safe, packing several transactions for Safe Multisend, etc.).
- **TypeHashHelper** - Helper library containing functions to build EIP712 struct and type hashes.
- **ConsoleOpBuilder** - Creates bytecode for common operations on Console Accounts: Enabling/Disabling a policy on a Console, Enabling/Disabling the Executor plugin on a Sub Account.



Systemic Risks

As with most blockchain frameworks, the use of Brahma for fund management entails several risks that are inherent to the system:

AddressProvider System

The reliance on the AddressProvider registry with governance ability to replace any core contract in the system poses several risks:

1. **Security Risk** - One of Brahma's main selling points is the fact that users maintain full custody of their assets. This assurance, however, can be easily broken if the governance address (which according to the docs is currently controlled by Brahma team only) decides to change some core contracts maliciously. For example, the ExecutorRegistry and the PolicyRegistry can be replaced with malicious versions that "register" a malicious address as an Executor and an all allowing policy on user Sub Accounts granting them full control on the Sub Account.
2. **Stability Risk** - The ability of replacing key contracts in the system without any restrictions on interface or structure poses the risk of code breaking, due to interface or structural changes that are incompatible with other contracts using the changed contracts.

External/Third Party protocol risk

As with any smart contract system, the reliance on third party code and packages might incur additional risks residing in these packages. With regards to Brahma, there is a significant dependence on Gnosis Safe as well as OpenZeppelin and Solady.

Offchain/External elements

Brahma's security approach places significant weight on its Policy system which relies on an external validator to sign policy validations thus enforcing user policies. Attention should be paid to serveral aspects of this mechanism with regards to security:

1. **Transparency** - As of the time for writing, we could not find any documentation of the policy enforcing mechanism, its code, available policy options or level of security. These are crucial to increase trust in the system.
2. **Availability** - Since the trusted validator is a crucial part of the system (without which users will have limited access to their safes), the availability guarantees of this component need to be made clearer.

Centralization Risks

Governance

As mentioned above, the system and its security rely entirely on the governance address (currently fully controlled by Brahma team). To increase trust and security, it is advisable to expand the spread of control; possibly by adding tiered governance, where for critical actions an additional governance address is required, including trusted entities external to Brahma, or even a community DAO.

Recommendations

Based on our review and the above-mentioned risks, we compiled the following list of recommendations that might address some of these risks:

Using Zero Knowledge Proofs for policy validation

In the current system, trust must be put in an external Trusted Validator to validate Policies which are central to Brahma's security framework. While the validator includes a relevant Policy hash in its signature to specify which policy it validated, there is no guarantee that the signed policy was indeed enforced. If the trusted validator is compromised, anyone can provide valid signatures without enforcing the policy. In the future, this trust assumption may be alleviated if a mechanism is integrated where policy validations are precompiled as ZKPs and are submitted together with a proof that the right policies have been enforced.

Employing a more robust and detailed governance system

As per current documentation, Brahma's governance address is a multisig of 10 addresses from within the team. In the future, it may be advisable to further decentralize governance by adding external signers (such as independent trusted industry figures) or even a DAO.

Limiting Wallet Registration to Brahma Enabled Console Safes

Currently, the system enables any address to be registered as a wallet and deploy Sub Accounts. It is recommended to limit this option to Brahma console accounts only. This is more aligned with Brahma protocol semantics and will prevent the option that users will grant a single EUA address full control of a sub safe by mistake.

Replacing the Current AddressProvider System With a More Standard Contract Updrage Mechanism

With the current system, core system contracts can be replaced at will without any backward compatibility checks, version controls, interface support or an option to set multiple new contracts as a batch. This might result in code breaks or unexpected system behaviour, which in turn, can result in security breaches or denial of service to safes. In the future, it may be beneficial to change this system (for example, with a standard `contractUpgradability` framework) or enhance it with more capabilities preventing the above-mentioned errors.

Time spent

15 hours

0xad1onchain (Brahma) acknowledged

0xsomeone (judge) commented:

> The analysis laid out in this report is exceptional; multiple diagrams have been created to showcase protocol flows as well as features. I find the Zero-Knowledge Proof-based recommendation slightly out-of-scope.

Disclosures

C4 is an open organization governed by participants in the community.

C4 Audits incentivize the discovery of exploits, vulnerabilities, and bugs in smart contracts. Security researchers are rewarded at an increasing rate for finding higher-risk issues. Audit submissions are judged by a knowledgeable security researcher and solidity developer and disclosed to sponsoring developers. C4 does not conduct formal verification regarding the provided code but instead provides final verification.

C4 does not provide any guarantee or warranty regarding the security of this project. All smart contract software should be used at the sole risk and responsibility of users.