SphereX Audit

OPENZEPPELIN SECURITY | AUGUST 9, 2023

Security Audits

August 9, 2023

This security assessment was prepared by **OpenZeppelin**.

Table of Contents

Summary

Type
: Security

Timeline
: From 2023-05-29
: To 2023-06-05

Languages
: Solidity

Total Issues
: 23 (14 resolved, 3 partially resolved)

Medium Severity Issues
5 (2 resolved, 2 partially resolved)
Low Severity Issues
12 (7 resolved, 1 partially resolved)
Notes & Additional Information
3 (3 resolved)

Scope

We audited the spherex-collab/spherex-protect repository at the 5b134b2b176deb94e75417ed9b9d3670444a4167 commit.

In scope were the following contracts:

```
src
├── SphereXEngine.sol
├── SphereXProtected.sol
├── Ownable.sol
└── ISphereXEngine.sol
```

System Overview

The SphereX-Protect system is designed to provide protection for smart contracts deployed on EVM blockchains. It consists of two main contracts: `SphereXEngine` and `SphereXProtected`.

`SphereXEngine` is the primary logic module containing protection rules, while `SphereXProtected` is an abstract contract utilized by the SphereX clients to integrate the monitoring mechanism into their contracts.

The SphereX-Protect system operates at the transaction level. During the execution of a transaction, data is sent from one or multiple contracts within the SphereX clients' smart contracts to the `SphereXEngine` contract. If a transaction trace (pattern) is not specifically allowed by the `SphereXEngine`, the behavior is classified as "suspicious", and the transaction is reverted.

behavior. Functions that are not decorated with these modifiers will not have their information collected or checked. Some specific integration guidance for using `SphereXProtected` in client contracts is provided in a separate section below.

***Update:** During the fix-review process, the system architecture was updated to add `SphereXProtectedBase` contract, and remove the `Ownable` contract.*

Call-Flow and Transaction-Flow

In the SphereX-Protect system, the behavior analysis is based on the concepts of Call-Flow and Transaction-Flow.

Call-Flow: Call-Flow represents the sequence of function calls within a program. In the context of SphereX-Protect, it refers to the collection of function calls made during the execution of a transaction. Each function call is assigned a unique positive integer ID, and the flow is represented as a vector. As functions are called, their IDs are added to the vector, and when functions return, the negative value of their IDs is added. By analyzing the Call-Flow, the system can detect deviations from expected patterns, indicating potential attacks.

Transaction-Flow: Transaction-Flow builds upon the concept of Call-Flow but considers the flow of multiple external calls within a single transaction. It provides more detailed information about the sequence of function calls within the transaction, as each external call is treated as part of one large vector. This allows for more accurate analysis.

The `SphereXEngine` contract offers both Transaction-Flow and Call-Flow. Both variations employ prefix checks to determine the validity of a flow. The choice between the two variants can be configured in the `SphereXEngine` contract.

Security Model and Trust Assumptions

SphereX-Protect operates on a data-driven model of trust, relying on collected transaction data to classify behavior as normal or suspicious. This data is collected during the transaction execution through a set of uniquely implemented modifiers, providing a dynamic and real-time basis for security decisions.

address, essentially controlling the data flow, while the latter identifies the endpoint for data transmission and transaction classification.

`SphereXEngine`, the brain of the system, houses crucial state variables that define its security logic: `_engineRules`, `allowedSenders`, and `allowedPatterns`. These variables specify the active security rules, the addresses permitted to send requests to the engine, and the transaction behaviors deemed acceptable, respectively.

Update: *During the fix-review process, the system was updated to separate the admin and the operator roles. This enables the admin role for both contracts to be controlled by the customer project. The operator role is intended to be controlled by the SphereX team, unless reassigned by the customer.*

Privileged Roles

In the frozen-commit audited, the project has two primary privileged roles:

1. **SphereXEngineOwner**: This role controls the main logic module, `SphereXEngine`, setting up protection rules, and managing the active state of the contract. The owner can add or remove `allowedSenders` and `allowedPatterns`, essentially deciding which contracts and behaviors the engine serves and permits.

2. **SphereXAdmin**: This role is in charge of the `SphereXProtected` contract, capable of changing the `SphereXEngine` address, and transferring the admin role to another address. As the control point for data flow from client contracts, this role is pivotal to the functionality of the SphereX-Protect system.

Given the powers associated with these roles, SphereX-Protect users should maintain the ownership of at least the `SphereXProtected` admin role, as a malicious admin can compromise contract security by redirecting to an engine that always reverts.

Update: *During the fix-review process, the system was updated to separate the admin and the operator roles. This enables the admin role for both contracts to be controlled by the customer*

Guidelines for Integration Audits

During future audits of customer projects integrating with this system, each integration should undergo a thorough examination with an emphasis on known issues, including the ones identified in this audit. The following is a non-exhaustive list of such issues:

- Call depth drift may be induced by arbitrary calls invoking a hook, or an inline assembly return occurring within an instrumented method's execution path.
- Correct application of modifiers:
  - Protected methods should not be invoked within loops, specifically, in Control Flow (CF) mode from another protected method, and Transaction Flow (TF) mode in general.
  - The `num` parameters in guard modifier arguments should be unique and positive.
- In TF mode, false positives could arise if third-party contracts call methods within loops or if either the third-party contract or the protected contract implements MultiCall functionality.
- The Engine's `_currentBlockOriginHash` should be checked to invalidate correctly on the target deployment chain, ensuring transactions are distinctly separated to minimize false positives.
- The `SphereXProtected` contract should correctly transfer ownership using `__SphereXProtected_init` when protocol ownership undergoes changes.
- Any logic that depends on revert messages from try/catch blocks or low-level calls won't be recorded and verified by the system.
- Only guard modifiers from `SphereXProtected` should be utilized, and not the hooks themselves.
- Systems built on Factory, Beacon, or personal proxy patterns should implement a mechanism to add each new contract instance as an allowed sender.

Update: *During the fix-review process,* `_currentBlockOriginHash` *was renamed to* `currentTxBoundaryHash` *and* `__SphereXProtected_init` *to* `__SphereXProtectedBase_init`.

Mitigation Recommendations for Client-Reported Issues

Ownership Transfer Omitted During Initialization

integrating contract might be changing.

Consequently, if the previous owner fails to invoke `changeSphereXAdmin` to transfer ownership, the former address retains its ownership rights. This could pose a security threat if this address becomes compromised, for instance, if it's a hot wallet deployer address. In such a situation, the entity controlling that address could permanently disable the Client contract by altering the Engine address using `changeSphereXEngine` to an address that reverts all calls to it (e.g., a non-zero EOA address).

Consider modifying the `__SphereXProtected_init()` function to always expect an owner address, to mitigate this potential risk.

***Update**: Resolved in pull request #27 at commit 052bb05.*

Assembly Return and Arbitrary Calls Can Cause Call Depth and Pattern to Drift

Inline assembly returns can interrupt the execution of the Post hook. Simultaneously, an arbitrary call can trigger the Pre hook on the Engine without invoking the corresponding Post hook. These inconsistencies break the invariant of calling both hooks and can result in a drift of the call depth counter and the pattern hash.

Consider implementing instrumentation for any potential arbitrary call site. This will ensure that the check executed during the Post hook in the guard modifier will cause the transaction to revert if there is any drift.

Also, the use of inline assembly returns should be scanned for and prohibited from being used in code callable from a protected method.

Finally, consider resetting the call depth and pattern at the transaction boundary for both modes of operation. This adjustment will limit the potential drift to a single transaction.

***Update**: Resetting the call depth was implemented in pull request #6 at commit c95fce7.*

Additional "Do No Harm" Principles

hold true:

> 1. `SphereXEngine` is immutable. Even in extreme cases of a malicious operator of `SphereXEngine`, the maximal potential damage would be limited to reverting transactions for the protected contracts. In that case, `SphereXEngine`'s Admin can renounce the malicious operator's ownership and restore control.
> 2. `SphereXProtected` does not collide with existing standard libraries/contracts (e.g., we do not store `SphereXAdmin` on slot 0 to prevent collision with OpenZeppelin Ownable).
> 3. The product does not alter the customer's contract logic. It only approves or denies transactions.
> 4. Our product cannot alter the state of the customer's contract (apart from predetermined `SphereXProtected`-related storage slots).
> 5. The engine does not censor transactions based on addresses (`tx.origin`, `msg.sender`).
> 6. Customer's or owner's funds are not processed by or stored inside `SphereXEngine` at any phase during or after execution of tx.
> 7. The `SphereXAdmin` is always capable of disabling the `SphereXEngine`, and therefore `SphereXEngine`'s owner is incapable of a DoS or ransom scenario.
> 8. `SphereXAdmin` has no control over the customers' contracts except for: (a) enabling/updating the engine, (b) disabling the engine, and (c) changing `SphereXAdmin`'s address.
> 9. Management functions: no one besides `SphereXAdmin` and `SphereXEngine`'s owner can access management functions for the `SphereXProtected` and `SphereXEngine` (respectively).

Risks concerning these principles were considered in our analysis. Although most of the principles hold, there are some important considerations for a few of the principles above:

- Regarding principle 5: Some censorship by `SphereXEngine` is discussed in finding M-05.
- Regarding principle 7: Short-term DoS risks caused by malice or false positives are discussed in findings H-03 and M-01.

engine can disable the customer's contract.

Update: *The section above was edited following the fix-review process to reflect the updated state of the codebase, as multiple findings that were related to principles 5, 7, 8, and 9 were resolved.*

High Severity

Call Depth Is Never Explicitly Reset

The Engine does not explicitly reset the call depth, which can lead to an off-by-one error in certain situations:

- If an arbitrary call to the Pre-hook is allowed from an unprotected method in an approved sender (a known client-reported edge case)
- In the event of a malicious or incorrectly implemented call to the Pre-hook from a contract mistakenly or maliciously added to approved senders
- If an assembly return occurs (another known client-reported edge case)

In scenarios where checks are bypassed due to `forceCheck` being set to false, the pattern could drift into an unpredictable state because it will not be reset. This could result in the rejection of all ensuing transactions once a check is eventually enforced during a Post-hook invocation (triggered from an external method).

Given its potential to cause a Denial-of-Service (DoS), it will require deactivating the protection mechanism.

Consider introducing explicit recording and invalidation of `currentBlockOriginHash` for both call flow and transaction flow cases, along with a reset of the call depth when `currentBlockOriginHash` is invalidated. This approach is valid because a call or a transaction cannot extend over multiple transactions, allowing each new transaction to always start from a depth of 1. In this manner, any drift scenario would only persist for the duration of that transaction instead of indefinitely. Furthermore, it is recommended to include all three scenarios in the testing suite to ensure any state corruption is confined to a single transaction.

Update*: Resolved in pull request #6 at commit c95fce7.*

logics are mutually exclusive. Specifically, one mode resets with each new transaction, while the other resets at each external call. The existing documentation further indicates that these modes are intended to operate individually rather than concurrently.

Consider adopting an Enum implementation for the modes to enforce exclusivity. Alternatively, input validation can be added on mode setters to prevent invalid configurations.

***Update**: Resolved in pull request #10 at commit d221420.*

Loop-Called Methods and MultiCall Contracts Require an Individual Allowance of All Likely Loop Lengths and Combinations

The current implementation requires explicit permission for each possible loop length if a protected method (internal, public, or external) is invoked within a loop. Furthermore, for contracts using the common MultiCall functionality, every likely combination of called methods will need to be allowed separately. This can result in false positives under these conditions:

- For Control Flow (CF): if a protected method is ever called from another protected method within a loop.
- For Transaction Flow (TF): in general, even without being called from another protected method.

For TF, this means that a third-party (e.g., an integrating) contract calling any protected method in a loop, or using a MultiCall could experience reverts. This reduces the attractiveness of TF to some integrators, as a protocol cannot realistically prevent third-party integrators from calling its methods within loops or from using MultiCall.

For CF, this means that a protected method should avoid calling another protected method in a loop.

This issue is also applicable to recursion, although recursion is rarely used in high-level methods in Solidity.

Consider prohibiting the instrumentation of calls within loops in integrating protocols, and recommend against using TF for protocols likely to be called by third-party contracts.

> *each number of repetitions within a range discussed with the customer. We plan to invest further research resources to find a generic solution for the TF case.*

Medium Severity

`currentBlockOriginHash` Invalidation Issues Can Lead to False Positives

Although reported as a known issue, this can be particularly severe in some circumstances:

- On some L2s, such as Arbitrum, where `block.number` is the L1 block number, so it will remain constant for some time. Similarly for Optimism, and zkSync.
- In chains that rely on relayers and Account Abstraction (quite possibly zkSync).

This can cause intermittent false positives as the second transaction could be rejected for producing a disallowed pattern.

Consider adding `block.timestamp` (will help with Arbitrum and Optimism) into the hash. It is possible that `block.difficulty`, `tx.gasPrice`, `block.baseFee` can also help.

***Update**: Resolved in pull request #11 at commit 51e369f.*

Potential for Hidden Attack Patterns During the Recording Phase

The recording phase of the contract could potentially be manipulated by an attacker to obscure harmful patterns within a large number of benign ones. This could allow a malicious user to introduce an attack vector into the list of permitted patterns.

For example, if an attacker discovers a reentrancy vulnerability, they could perform numerous transactions, some of which involve harmless reentrancy (resulting in no loss of funds), with the hope of getting this pattern added to the allow-list.

Once the protection is activated and the total value locked (TVL) within the contract reaches a significant level, the attacker may exploit the allow-listed attack pattern.

Consider filtering the initially recorded patterns, particularly if there are a large number of them.

> *recording phase data (identifying abnormal transactions and analyzing them before approving them into the permissible patterns).*

Client Project Has No Ownership Over the Engine Configuration

At present, the contract engine is solely governed by a single administrator address. As this address is expected to actively operate the contract, it is most likely to be a Spherex-controlled address. This means that if SphereS, as the admin, fails to cooperate, the client project can only deactivate the protection mechanism or deploy its own engine and update its state. This creates an imbalanced relationship since the client has no direct control over patterns being accepted or rejected.

This issue's severity is exacerbated by a lack of emitted events during both configuration and operation, which would hinder the client from configuring their "forked" engine instance.

To address this, consider implementing Role-Based Access Control (RBAC). Under this framework, the client project would have admin control while SphereX is granted an operational role, enabling them to update patterns and other configurations. If the client project is dissatisfied with recent configuration updates, they can take over the operational role, reverse any changes, and maintain the previous protection level. Additionally, consider implementing events such that a community member would be able to reconstruct the state of the Engine with common tools.

***Update**: Resolved in pull request #14 at commit ec71add.*

No Events Emitted and Lack of External Views in Contracts

The contracts `SphereXProtected` and `SphereXEngine` do not emit events or implement external views. This absence of conventional observability methods demands specialized tools and knowledge to monitor and inspect the operational state of these contracts. Moreover, it impedes the utilization of existing data analysis tools, dashboards, monitoring utilities, and front-end frameworks. This opacity can diminish trust, neutrality, and the overall utility of the system for integrating projects.

This lack of observability can introduce further security risks. For instance, if the engine is deactivated by the admin, whether intentionally or unintentionally, it could go unnoticed due to

We recommend implementing comprehensive observability of all state changes using both views and events. The entire current state of the contract should be visible through views and reconstructible via events. Additionally, to facilitate gas savings for the customer, consider providing a feature to control event emission during hook execution, possibly through a configuration or input flag. This feature may be more relevant for deployments on Ethereum L1, as the extra gas cost for emitting events on L2s or sidechains should be insignificant.

***Update**: Partially resolved in pull request #15 at commit f0a818b. Configuration-related events were added. However, state changes involved in hook execution remain unaccompanied by events, which will make tracking the operation of the engine harder to track off-chain.*

Potential Issues in Systems Using Factory and Beacon Patterns

Various smart contract systems implement patterns that dynamically add contracts to the system based on user actions. Some notable examples include pool creation in AMMs, personal proxies in Maker, and multi-signature (multi-sig) Safe creation. When protected by SphereX, each newly created instance of a factory-created contract would need to be separately added by the admin, preventing the contract from becoming immediately operational upon creation, adding a centralizing step, and adding operational complexity.

Furthermore, performing changes to admin and engine addresses on the dynamically generated contracts will be operationally complex and error-prone.

Moreover, these systems can pose a risk of targeted censorship due to the capability of SphereX to exclude a specific personal proxy address from the allowed senders.

Consider implementing Role-Based Access Control (RBAC) as described in the OpenZeppelin documentation. One possible approach could be creating a `SENDER_CONFIG_ROLE` with the permissions to add allowed senders. This will allow the general admin to assign this role to the factory contract when needed. Alternatively, consider establishing an additional administrative function that enables a factory contract to independently add a newly created contract as an approved sender.

To alleviate the censorship risk, consider separating the storage of factory-added addresses and disallowing the removal of any specific address from that set. Alternatively, consider implementing a long delay for removing senders, such that a censored personal proxy owner will have time to exit the system if both SphereX and the Client project are forced to censor it.

***Update**: Partially resolved in pull request #27 at commit 052bb05. A role-based system was implemented, allowing factories to add new senders on-chain. The configuration and censorship concerns were not resolved.*

Low Severity

Outdated Solidity Versions

Throughout the codebase, there are `pragma` statements that use an outdated version of Solidity. For instance:

- The `pragma` statement on line 4 of `ISphereXEngine.sol`
- The `pragma` statement on line 4 of `SphereXEngine.sol`
- The `pragma` statement on line 4 of `SphereXProtected.sol`

Consider taking advantage of the latest Solidity version to improve the overall readability and security of the codebase by avoiding known bugs and taking advantage of newer language features.

Furthermore, it is highly unlikely that newly deployed customer contracts will use versions of Solidity that are older than 0.8. To accommodate specific clients who may require an older version, code can be back-ported and audited separately for that specific integration as part of the integration audit (which is needed for other reasons as well).

***Update**: Resolved in pull request #16 at commit 9b6c897.*

Possible Underflow Due to Unnecessary Signed Integers in Interfaces

`SphereXEngine` hooks lack validation of the sign of `num` parameter. This can cause an underflow when adding it to `_callDepth` in `_addCFElement` if using a Solidity version older

In addition to being error-prone, the use of `int16` in the signatures of `sphereXGuard` modifiers in `SphereXProtected` and hooks in `SphereXEngine` is redundant. This is because a positive number is always expected for the Pre hooks, and a negative, exactly opposite number is expected for the Post hooks. Thus, the negative sign can always be safely added in the Post hooks, leaving the engine's implementation encapsulated and separated from the interface.

Consider using `uint256` for the identifier in all the modifiers and the hooks, and using signed integers only in the internal calls in `SphereXEngine`. Alternatively, consider adding input validation to the hooks to ensure that the sign is as expected.

***Update**: Resolved in pull request #6 at commit c95fce7.*

Transaction Flow Mode Vulnerable to Circumvention Via Flashbot Bundles

A complex attack transaction (disallowed by the engine) involving multiple calls to the client contract may be subdivided into several transactions and executed in sequence, using a private mempool like flashbots. This may allow circumventing the protection of TF mode by employing different externally owned accounts (EOAs) as `tx.origin`.

For instance, an attack requiring a sequence of `[attacker: action_A, attacker: action_B]` could appear as a novel transaction flow pattern if executed from an attacking contract. To circumvent this, the attack could be submitted as a bundle of EOA transactions: `[attacker: action_A, decoy: action_C, attacker: action_B]`. Each transaction in this bundle would pass transaction flow validation as a separate flow. Although certain attacks, such as flashloan attacks, can't be fragmented in this manner, many other multi-step attacks can be.

Consider documenting this option as a consideration for correctly configuring the engine for the client's security needs.

***Update**: Acknowledged. The SphereX team stated:*

> *This possible bypass scenario is to be taken into account and will be the subject of future research.*

An integrating protocol, even one with an ostensibly limited control area, might exploit the `sphereXGuard` modifiers as a "backdoor" by transitioning to a custom engine implementation.

For instance, a protocol could introduce a custom engine that disallows non-insider users from interacting with it. This restriction could prevent users from managing their assets. This will enable insiders to extract these assets by exploiting protocol-specific mechanisms that depend on unrestricted access. Examples could involve using liquidations for lending protocols or arbitrage for trading protocols.

Consider documenting the trust assumptions of the system such as the expanded control surface and potential centralization risks that can arise, so that they are clear to the user communities of the integrating protocols.

***Update**: Acknowledged. The SphereX team stated:*

> *We will explicitly state this issue in the trust assumptions of the system. Moreover, we will state in our legal documents that we are not liable for any engine that was not deployed and operated by SphereX.*

Potential Risks with Hooks' Internal Visibility

The `sphereXGuard` modifiers are designed to be used by integrating protocols. However, the visibility of the `_sphereXValidatePre`, `_sphereXValidatePost`, `_sphereXValidateInternalPre`, and `_sphereXValidateInternalPost` methods is currently set to `internal`. This setting could allow an improperly implemented integration to call the engine methods directly without using the modifiers, thereby potentially breaking the invariant of invoking both hooks around each instrumented call.

To minimize the likelihood of erroneous integrations, consider changing the visibility of these methods to `private`.

***Update**: Resolved in pull request #19 at commit 1baed67.*

Unrecorded Calls Resulting from `try/catch` or Low-Level Reverts

This is due to the fact that they revert their changes in the Engine.

An illustrative example can be seen in the counterfactual, always-reverting rollback logic in the Synthetix V3 upgrade module. Here, calls always revert but with varying revert errors.

We recommend documenting this edge case to ensure it is considered during integration audits.

***Update**: Acknowledged. The SphereX team stated:*

> *We will document this edge case.*

Potential Loss of Ownership During Transfer

The `SphereXEngine` implements a single-step ownership transfer mechanism that is prone to operator errors.

Consider using the `Ownable2Step` contract from the OpenZeppelin library for a safer transition of ownership. This will increase the contract's flexibility and safeguard against unintended loss of control over the engine. Also, the inclusion of a renounce function should be considered to enable owners to more transparently renounce their ownership.

***Update**: Resolved in pull request #14 at commit ec71add.*

SphereXEngine Rule Configuration Methods Poorly Named

In the implementation of the SphereXEngine contract, the `activateRules` method allows the owner to deactivate the rules by inputting all 8 bytes as zero. This is counterintuitive as `activateRules` should logically serve to enable the rules, not deactivate them. The `deactivateRules` method, meant for rule suspension, becomes redundant as the same action can be performed through the `activateRules` method.

In order to avoid any misinterpretation or misuse, it is recommended to integrate a validation process within the `activateRules` method that prevents rule deactivation. Alternatively, one could consider consolidating these functions into a single method, `configureRules`, that toggles the state of the rules based on the input provided, thus eliminating any redundancy and making the system's interface more intuitive.

The `changeSphereXAdmin` method currently allows for a single-step ownership transfer, which can be risky due to potential mistakes leading to loss of control to update the Admin or the Engine address and thus losing the ability to opt out of the SphereX Protect.

Consider using a two-step ownership transfer to provide a safer mechanism. Moreover, the method should also validate that the proposed address conforms to the expected contract interface to ensure correct functionality. Lastly, the method should be made `virtual` to permit override by integrating contracts with their own access control measures.

***Update**: Resolved in pull request #14 at commit ec71add.*

Gas Inefficiencies

There are many instances throughout the codebase where changes can be made to improve gas consumption. For example:

1. Consolidate `_callDepth`, `_currentPattern`, and `_currentBlockOriginHash` into a single slot. This can be done using a Struct that defines variables that will fit into a single slot. For example, using `uint16` for `_callDepth`, `uint216` for `_currentBlockOriginHash`, and `bytes3` for `_currentBlockOriginHash`. This optimization reduces storage operations and may save up to 5000-7000 gas per transaction during execution on L1.
2. Avoid performing multiple storage reads and writes on `_callDepth`. `_currentBlockOriginHash` and `_currentPattern` during `_addCFElement` by reading once into stack variables, and writing back into storage at the end of the method.
3. Avoid checking for the active rule twice during `_addCFElement` by using a separate internal method for each case. Additionally, splitting the method into two distinct methods will make the code more readable, auditable, and testable. This will also make the exclusivity of the modes clear.
4. Use modifiers with internal methods. By doing so, bytecode size is reduced.
5. The `returnsIfNotActivated` check can be performed once in the `SphereXGuard` modifiers, rather than being checked twice for each hook (Pre and Post).

7. Use a more recent version of Solidity to take advantage of compiler gas optimizations for both deployment and execution costs.
8. Use custom errors implementation (if upgrading to a more recent version of Solidity) to save on deployment cost and failed execution gas cost.

When performing these changes, aim to reach an optimal tradeoff between gas optimizations and readability. Having a codebase that is easy to understand reduces the chance of future errors and improves community transparency.

***Update**: Partially resolved. Suggestion 1 was implemented in pull request #23 at commit 950a31d. Suggestions 2 and 3 were implemented in pull request #6. Suggestion 7 was implemented in pull request #16. Regarding the other suggestions, the SphereX team stated:*

> *After deeply discussing every gas inefficiency issue, we have reached an optimal tradeoff from our point of view.*

SphereXEngine Migration Flow Susceptible to Human Error

The `SphereXEngine` migration process is currently prone to errors if it involves multiple smart contracts using `SphereXProtected`. When `SphereXEngine` requires a change, the admin of each contract must manually replace the engine address. This approach becomes problematic if any contracts are inadvertently overlooked and continue pointing to an outdated version of the `SphereXEngine`. This can be particularly problematic, or operationally complex when dealing with a large or dynamic system of contracts (e.g., contracts using a factory pattern).

Consider introducing a periphery contract that enables updating all associated contracts in a single transaction. Alternatively, consider implementing a pattern similar to the beacon proxy pattern. The beacon proxy pattern centralizes the update mechanism and reduces the manual workload. By referencing a single 'beacon' that holds the updated `SphereXEngine` address, all contracts could automatically reflect the most recent engine version, thus reducing the risk of misalignment and potential system malfunctions.

***Update**: Acknowledged, not resolved. The SphereX team expressed that a separate config contract is not desirable due to gas costs.*

confuse users of the integrating protocols due to their lack of descriptiveness. It is recommended to revise these messages to more accurately depict the corresponding issues and context (e.g., `"SphereX error: disallowed tx pattern"`). This change will enhance user understanding and facilitate troubleshooting of contract interaction issues.

***Update**: Resolved in pull request #22 at commit b533a0b.*

Notes & Additional Information

Non-Explicit Imports Are Used

The use of non-explicit imports in the codebase can decrease the clarity of the code, and may create naming conflicts between locally defined and imported variables. This is particularly relevant when multiple contracts exist within the same Solidity files or when inheritance chains are long.

Throughout the codebase, global imports are being used. For instance:

- Line 6 of `SphereXEngine.sol`
- Line 7 of `SphereXEngine.sol`
- Line 6 of `SphereXProtected.sol`

Following the principle that clearer code is better code, consider using named import syntax (`import {A, B, C} from "X"`) to explicitly declare which contracts are being imported.

***Update**: Resolved in pull request #20 at commit cfdd63c.*

Usage of Small-Size Variables

The usage of `int16` and other small number types is present in several parts of the codebase. It is advisable to refrain from using these types in memory calls, interfaces, and calculations. Only employ them when writing into storage for storage slot packing. Utilizing these smaller size variables can lead to potential issues:

- Overflows and underflows in mathematical and casting operations can remain undetected
- Increased gas expenditure during execution and larger contract bytecode size
- Reduced code readability

sizes. Perform downcasting (safely) only when writing values into storage.

***Update**: Resolved in pull request #18 at commit 0df3f47.*

Typographical Errors

There are typographical errors in the codebase's comments. For example:

- "Poistive" should be "Positive".
- "insturction" should be "instruction".
- "nody" should be "body".
- "rotected" should be "protected".
- "defence" should be "defense".
- "abitrary" should be "arbitrary".
- "visibality" should be "visibility".
- "defenitions" should be "definitions".
- "Technolegie"s should be "Technologies".

Consider scanning the codebase using automated tooling and correcting typographical errors.

***Update**: Resolved in pull request #21 at commit ed86e8a.*

Three high-severity issues were found among various lower-severity issues. The SphereX team's documentation, diagrams, and examples aided the auditors in evaluating and understanding the code. The team promptly addressed questions and suggestions raised by the auditors throughout the audit.

***Update**: During the fix-review process, a significant amount of issues were resolved, including two high-severity issues.*

Monitoring Recommendations

With SphereX emerging as a ground-breaking Web3 technology, comprehensive monitoring strategies are integral to maintaining its integrity and security. This section recommends practical and actionable measures that would enhance the reliability and transparency of SphereX systems:

- Modifications made to the SphereX engine address on each smart contract that uses the `SphereXProtected` contract.
- Changes to the admin role of each `SphereXProtected` smart contract.
- Activation and deactivation of rules in the `SphereXEngine` contract.
- Addition or removal of allowed patterns. The introduction of a malicious pattern could render the engine incapable of protecting the contract from an attack. Conversely, an unnotified removal could create a Denial-of-Service (DoS).
- Additions or removals of sender addresses that interact with the `SphereXEngine`. Changes may represent potential threats or anomalies that need immediate attention.

Implementing these monitoring recommendations will significantly enhance the ability to quickly detect and respond to any potential issues, thereby improving the overall security and performance of SphereX.

Related Posts

intermediary designed to execute users' orders through routes...

Security Audits

OpenBrush is an open-source smart contract library written in the Rust programming language and the...

Security Audits

Ethereum. It is designed to be EVM-compatible and aims to...

Security Audits