About Trail of Bits

Founded in 2012 and headquartered in New York, Trail of Bits provides technical security assessment and advisory services to some of the world's most targeted organizations. We combine high-end security research with a real-world attacker mentality to reduce risk and fortify code. With 100+ employees around the globe, we've helped secure critical software elements that support billions of end users, including Kubernetes and the Linux kernel.

We maintain an exhaustive list of publications at https://github.com/trailofbits/publications, with links to papers, presentations, public audit reports, and podcast appearances.

In recent years, Trail of Bits consultants have showcased cutting-edge research through presentations at CanSecWest, HCSS, Devcon, Empire Hacking, GrrCon, LangSec, NorthSec, the O'Reilly Security Conference, PyCon, REcon, Security BSides, and SummerCon.

We specialize in software testing and code review projects, supporting client organizations in the technology, defense, and finance industries, as well as government entities. Notable clients include HashiCorp, Google, Microsoft, Western Digital, and Zoom.

Trail of Bits also operates a center of excellence with regard to blockchain security. Notable projects include audits of Algorand, Bitcoin SV, Chainlink, Compound, Ethereum 2.0, MakerDAO, Matic, Uniswap, Web3, and Zcash.

To keep up to date with our latest news and announcements, please follow @trailofbits on Twitter and explore our public repositories at https://github.com/trailofbits. To engage us directly, visit our "Contact" page at https://www.trailofbits.com/contact, or email us at info@trailofbits.com.

Trail of Bits, Inc.
228 Park Ave S #80688
New York, NY 10003
https://www.trailofbits.com
info@trailofbits.com

Notices and Remarks

Copyright and Distribution

Test Coverage Disclaimer

All activities undertaken by Trail of Bits in association with this project were performed in accordance with a statement of work and agreed upon project plan.

Security assessment projects are time-boxed and often reliant on information that may be provided by a client, its affiliates, or its partners. As a result, the findings documented in this report should not be considered a comprehensive list of security issues, flaws, or defects in the target system or codebase.

Trail of Bits uses automated testing techniques to rapidly test the controls and security properties of software. These techniques augment our manual security review work, but each has its limitations: for example, a tool may not generate a random edge case that violates a property or may not fully complete its analysis during the allotted time. Their use is also limited by the time and resource constraints of a project.

Table of Contents

Executive Summary

Engagement Overview

The Organisation engaged Trail of Bits to review the security of the Synapse Protocol smart contracts. The Synapse Protocol is an optimistic cross-chain bridge protocol for sending general messages from one blockchain to another blockchain.

Our testing efforts focused on finding issues impacting the availability and integrity of the target system. With full access to source code and documentation, we performed static and dynamic testing of the Synapse Protocol smart contracts using automated and manual processes. The off-chain software components used by the system were not in scope for this review.

Observations and Impact

The Synapse Protocol smart contracts are divided into well-defined components with limited responsibility. The smart contracts are designed to be deployed on multiple blockchains, with a focus on minimizing code duplication. However, the complex inheritance chain of contracts, where base contracts have functions for multiple components on different chains, makes it difficult to follow the code path and understand the security properties of all the system components.

During the review, we found one high-severity issue, TOB-PSC-03, arising from insufficient security measures for detecting fraud in the system's optimistic execution flow. We also found several issues related to poor incentive alignment for privileged system actors (TOB-PSC-06, TOB-PSC-11, TOB-PSC-13, TOB-PSC-16, and TOB-PSC-18) .

Some issues, such as TOB-PSC-05, TOB-PSC-08, TOB-PSC-15, and TOB-PSC-16, are the result of using vulnerable mechanisms for passing crucial data like the `agentRoot` and `gasData` parameters from one blockchain to another blockchain. Other issues are related to arithmetic errors, such as integer overflow, unsafe casting, and rounding behavior.

Recommendations

Based on the codebase maturity evaluation and findings identified during the security review, Trail of Bits recommends that the team take the following steps:

- **Remediate the findings disclosed in this report.** Various issues allow privileged actors to execute malicious messages and impact system availability. These findings should be addressed as part of a direct remediation or as part of any refactor that may occur when addressing other recommendations.

- **Review the alignment of incentives for system actors.** To function properly, the protocol relies on guards and notaries to perform their duties. They are incentivized to work honestly through rewards, staking, and slashing mechanisms. The improper incentive alignment introduces several security risks impacting the system's liveness and integrity. To discourage malicious activity, we recommend that the team carefully review the reward amounts and beneficiaries, staking amounts, and slashing mechanisms and ensure that actors doing honest work are rewarded and that malicious actors are punished.

- **Reduce the system's overall complexity.** The interactions between different system components are complex and span several different technology stacks with different security assumptions and guarantees. This makes it difficult for any one developer to understand the security properties provided by the different components. We recommend reducing the system complexity by segregating components with respect to the usage domain and with explicit use of applicable domain names in the function names. For example, fraud detection, dispute raising, and dispute resolution functions are shared by contracts deployed on all blockchains and are implemented in the base contracts. These functions can be renamed to include domain qualifiers such as `origin`, `destination`, `synapse`, and `common` to explicitly designate their usage domain.

- **Expand existing documentation.** The Synapse Protocol would benefit from a system-level specification with a focus on valid states and state transitions. Such a specification will help identify paths leading to inconsistent states and avoid security issues arising from invalid state transitions.

- **Expand the existing testing suite.** In addition to existing unit and integration tests, the team should consider implementing system invariants and fuzzing them with Echidna. Invariant testing will help secure the system by ensuring conformity of system properties.

The following tables provide the number of findings by severity and category.

EXPOSURE ANALYSIS

Severity	*Count*
High	7
Medium	6
Low	3
Informational	6
Undetermined	0

CATEGORY BREAKDOWN

Category	*Count*
Auditing and Logging	1
Configuration	1
Data Validation	14
Denial of Service	2
Patching	1
Timing	3

Project Goals

The engagement was scoped to provide a security assessment of the Synapse Protocol. Specifically, we sought to answer the following non-exhaustive list of questions:

- Can a user stop the bridge for an origin or destination blockchain?
- Can a user execute a malicious base message or manager message on a destination chain?
- Can a guard or notary steal tips?
- Can users lose money by using the bridge smart contracts?
- Can some assets become stuck in the bridge smart contracts?
- Can a user bypass the system's access control checks?
- Can a guard stop notaries from performing their duties?
- Can a notary stop guards from performing their duties?
- Can a user stop guards or notaries from performing their duties?
- Can a user, guard, or notary stop the owner from performing their duties?
- Can a user, guard, or notary stop other users from sending messages through the bridge?
- Can an executor benefit from message execution reordering on the destination chain?
- Can a notary manipulate the `GasOracle` contract to earn higher tips?
- Can a user cause loss to guards or notaries by sending non-executable messages?
- Can an unexpected state update go unnoticed by the monitoring system?
- Are all the privileged actors incentivized to perform their duties honestly?
- Can a message be replayed?

Project Targets

The engagement involved a review and testing of the following target.

Synapse Protocol smart contracts

Repository	https://github.com/synapsecns/sanguine
Version	b9a87f6c7bf51335dcbc83092f8135be091cda36
Directory	/packages/contracts-core/contracts
Type	Solidity
Platform	Ethereum

Project Coverage

This section provides an overview of the analysis coverage of the review, as determined by our high-level engagement goals. Our approaches included the following:

- **General.** We reviewed the entire codebase for common Solidity flaws, such as missing contract existence checks on low-level calls, issues with access controls, unimplemented functions, and issues related to upgradeable contracts, such as state variable storage slot collision and initialization front-running. We used Slither to statically analyze the code and Echidna to fuzz test system invariants.

- **Message passing flow.** We reviewed how messages are passed through the system, verified by agents, and executed. We ensured messages cannot be replayed on the same blockchain or a different blockchain, checked that system agents cannot execute fraudulent base messages or manager messages by circumventing the expected flow of execution, and reviewed risks related to front-running and reordering of messages.

- **Fraud detection, dispute, and slashing mechanisms.** The Synapse Protocol is an optimistic message-passing bridge and uses a fraud detection mechanism to stop the execution of malicious messages through the protocol contracts. The protocol assigns guards to monitor notary activity. The guards validate every snapshot, attestation, and receipt submitted by the notaries and raise a dispute upon detecting an invalid one. The guard or notary can then be slashed by proving the correctness of the snapshot, attestation, receipt, or report. We reviewed the event emissions and state verification functions used to facilitate fraud detection. We also looked at the impact of an agent being disputed only once and how this limitation can be exploited by agents to execute malicious messages. Finally, we reviewed the delay enforced for the execution of messages, signature malleability issues, and incentive alignment for agents to work honestly, ensuring that agents cannot halt the bridge or grieve users.

- **Tip collection and distribution.** The Synapse Protocol charges tips for sending base messages from one blockchain to another. These tips are used to incentivize the guards, notaries, and executors to verify, propagate, and execute, respectively, the messages on the destination blockchain. Once a message is executed, the notary can submit a message receipt to the `Inbox` contract on the Synapse chain to add it to the receipt queue. All receipts are processed, and the tip values they contain are used to increase the agent's internal balances, which they can withdraw by initiating the withdrawal on the Synapse chain's `Summit` contract. We reviewed how receipts are submitted and processed, the way tips are calculated and whether this calculation can be truncated or rounded down to zero, and how notaries can affect the tip values by submitting attestations. Additionally, we reviewed whether tips can become stuck in the contracts, whether agents can claim tips for a receipt multiple times, whether agents can claim tips without doing the required work, or whether any user can inflate the value of their tips or steal other users' tips. Finally, we looked at the possibility of agents increasing the tips to stop users from sending messages through the bridge and ensured that users cannot send messages without paying the required tips.

- **`GasOracle` contract.** The `GasOracle` contract stores the current gas price information of all available destination blockchains. The current gas price is used to calculate the minimum amount of tips a user must provide to send a base message to a destination blockchain. The gas price information can be set by the contract owner, or it can be updated via notary-provided attestations. We reviewed how the gas data is updated in the `GasOracle` contract to ensure that it cannot be manipulated by agents to benefit from tips or affect the bridge's liveness and integrity. We also looked at the arithmetic operations to analyze the impact of rounding behavior and approximations made in the calculations. Finally, we reviewed the possibilities and impact of the `gasData` variable getting out of sync with the actual gas parameters of the remote blockchains.

- **Merkle tree library contracts.** The Merkle tree library contracts implement the incremental Merkle tree and dynamic Merkle tree structures used to store messages and agents in an easily verifiable way. We reviewed the Merkle tree library contracts to ensure that the Merkle tree proofs cannot be faked by changing the index of the leaf node or the length of the proof. We looked at the hashing schema of the leaf nodes and the intermediate nodes to ensure that using the same hashing schema for all the nodes did not introduce any vulnerabilities. Finally, we also looked at the impact of using zero value instead of zero hashes for empty nodes in the incremental Merkle tree.

Coverage Limitations

Because of the time-boxed nature of testing work, it is common to encounter coverage limitations. The following list outlines the coverage limitations of the engagement and indicates system elements that may warrant further review:

- **Off-chain components.** The Synapse Protocol uses off-chain components to relay messages from one blockchain to another blockchain. These components were out of scope, so they were not reviewed.

- **Deployment scripts.** The code directory in scope does not include the deployment scripts, so they were not reviewed.

Automated Testing

Trail of Bits uses automated techniques to extensively test the security properties of software. We use both open-source static analysis and fuzzing utilities, along with tools developed in house, to perform automated testing of source code and compiled software.

Test Harness Configuration

We used the following tools in the automated testing phase of this project:

Tool	Description	Policy
Slither	A static analysis framework that can statically verify algebraic relationships between Solidity variables	Appendix D
Echidna	A smart contract fuzzer that can rapidly test security properties via malicious, coverage-guided test case generation	Appendix D

Test Results

The results of this focused testing are detailed below.

General system properties. Using Echidna, we tested the following system properties.

Property	Tool	Result
If the preconditions are met, the `execute` function never reverts.	Echidna	TOB-PSC-14
Messages that have been successfully executed cannot be executed again.	Echidna	**Passed**
The `isValidReceipt` function always returns `true` for valid receipts.	Echidna	**Passed**
The `isValidReceipt` function always returns `false` for invalid receipts.	Echidna	**Passed**

If a new agent is added or an existing agent is updated, the `agentRoot` is always updated.	Echidna	**Passed**
An agent's domain can never be changed.	Echidna	**Passed**
If the preconditions are met, adding or updating an agent always succeeds.	Echidna	**Passed**
An agent cannot be added multiple times.	Echidna	**Passed**
An agent cannot be slashed multiple times.	Echidna	**Passed**
A disputed agent cannot dispute again.	Echidna	**Passed**
An agent that has been slashed cannot become active again.	Echidna	**Passed**
The `isValidAttestation` function always returns `true` for valid attestations.	Echidna	**Passed**
The `isValidAttestation` function always returns `false` for invalid attestations.	Echidna	**Passed**
Sending a base message should never revert as long as the preconditions are met.	Echidna	**Passed**
As long as the preconditions are met, agents can be put in dispute.	Echidna	**Passed**
As long as the preconditions are met, agents can be slashed.	Echidna	**Passed**

Codebase Maturity Evaluation

Trail of Bits uses a traffic-light protocol to provide each client with a clear understanding of the areas in which its codebase is mature, immature, or underdeveloped. Deficiencies identified here often stem from root causes within the software development life cycle that should be addressed through standardization measures (e.g., the use of common libraries, functions, or frameworks) or training and awareness programs.

Category	Summary	Result
Arithmetic	The codebase uses arithmetic operations to calculate tips, with inline documentation of approximations and test cases covering the operations. However, the system lacks a systematic approach toward explicit rounding behavior. Applying the recommendations specified in our rounding recommendations in appendix F and using automated testing tools like Echidna would help mitigate risks and improve the evaluation of the system's loss-of-precision risks.	Moderate
Auditing	Critical state-changing functions, such as updating the `summitTip` value in the `GasOracle` contract, do not emit events. The documentation does not indicate how events are monitored off-chain to detect abnormal behavior. Additionally, no incident response plan has been prepared, which would make the recovery process difficult in the event of a compromise (see appendix G).	Weak
Authentication / Access Controls	The system implements only a few access control roles for privileged functions, in accordance with the principle of least privilege. However, the system lacks a two-step process for crucial privileged operations, and the existing test suite does not cover the access controls of all the privileged functions. The system can benefit from additional documentation specifying the roles, their privileges, and the process of granting and revoking the roles in the system.	Moderate
Complexity Management	In general, the code is well structured with small functions that are easy to test. However, the contracts	Moderate

	have a complex inheritance tree, with base contracts having functions for different components deployed on the different chains. The functions with undefined scopes in base contracts make it difficult to understand the entire code flow and the security properties of the different system components. The system can benefit from segregating components based on their usage domain. Similarly, explicitly designating the scope of functions in the base contracts would help reduce the complexity of the codebase for the maintainer.	
Decentralization	Synapse Protocol is designed to be a decentralized protocol. The owner can change system parameters at any time, and agents can be added only by the owner. However, the system does not implement a timelock mechanism for updating crucial system parameters, which can impact users and decentralized applications (dapps) integrating with the protocol. Additionally, all contracts are upgradeable by the team, and the dapps integrating with the bridge cannot opt out of parameter changes and upgrades. Implementing a governance system with a timelock for upgrading the contracts and updating important system parameters could benefit the system decentralization. Additionally, clearly documenting owner-controlled parameters and their impact on users, as well as implementing a permissionless way of adding agents, would help inform users and reduce owner control.	**Weak**
Documentation	The documentation is a multi-level specification of the important components. The codebase includes inline and NatSpec comments, and there is consistency across all documentation. However, additional documentation on the gas parameters and tip amount arithmetic and precision is required to ensure the correct use of admin functions updating the gas parameters and tips.	**Moderate**
Front-Running Resistance	User actions on the Synapse Protocol are resistant to general maximum extractable value (MEV) risks because users can send a message only from the origin chain and execute a message only on the destination chain. However, further work could be done to ensure admin, guard, and notary operations are protected from	**Moderate**

	front-running risks. Finally, the documentation and tests lack a focus on front-running risks.	
Low-Level Manipulation	The protocol extensively uses low-level operations. Extra documentation and testing are provided for low-level code. However, the system can benefit from a high-level reference implementation and automated tests validating the low-level code against it.	**Moderate**
Testing and Verification	The project has an extensive unit and integration test suite that tests both the happy path and various failure cases and adversarial inputs. An automated testing technique is used for critical components. However, the codebase would benefit from including property testing, which would help verify critical system invariants. Once the properties and invariants have been documented, they can be used as a baseline to improve the unit tests and, in the future, to include automated testing techniques such as fuzzing.	**Moderate**

Summary of Findings

The table below summarizes the findings of the review, including type and severity details.

ID	Title	Type	Severity
1	Attestation nonce overflow leads to wrong payment of snapshot tips	Data Validation	High
2	Initialization functions are vulnerable to front-running	Timing	Informational
3	Notaries can execute arbitrary messages to arbitrary receivers	Data Validation	High
4	Lack of upper bound on the summit tip	Data Validation	Informational
5	The bridge may not function for a destination chain	Configuration	High
6	Guards can grieve honest notaries by disputing them during peak traffic	Denial of Service	High
7	A portion of the message value gets locked in the Origin contract	Data Validation	Medium
8	Resolving stuck disputes can be indefinitely delayed	Data Validation	Medium
9	Risk of message execution reordering	Timing	Medium
10	Tip calculation can round down to zero	Data Validation	Informational
11	Agents can collude to execute malicious manager messages	Data Validation	High
12	Lack of a two-step process for ownership transfer	Data Validation	High

13	Slashing mechanism makes guards and notaries likely targets of phishing campaigns	Data Validation	**High**
14	Tips for messages with an EOA recipient cannot be claimed	Data Validation	**Medium**
15	Notaries can submit a high gas value to stop users from sending messages to a destination chain	Denial of Service	**Medium**
16	Lack of incentives to update gasData to a lower value	Data Validation	**Low**
17	Risks of executing a base message on a non-recipient contract	Data Validation	**Low**
18	Front-running risks in functions rewarding the caller	Timing	**Medium**
19	Use of outdated third-party dependencies	Patching	**Informational**
20	Insufficient event generation	Auditing and Logging	**Informational**
21	Unsafe cast can be problematic	Data Validation	**Informational**
22	Using msg.value in a loop is unsafe	Data Validation	**Low**

Detailed Findings

1. Attestation nonce overflow leads to wrong payment of snapshot tips	
Severity: **High**	Difficulty: High
Type: Data Validation	Finding ID: TOB-PSC-01
Target: `hubs/SnapshotHub.sol`, `inbox/Inbox.sol`, `Summit.sol`	

Description

The attestation nonce can overflow the upper limit of the `uint32`. This overflow leads to the wrong payment of the snapshot tips and unexpected behavior for off-chain components.

The `_saveNotarySnapshot` function in the `SnapshotHub` contract computes the attestation nonce by casting the length of the `_attestations` array to a `uint32` value:

```
function _saveNotarySnapshot(
    Snapshot snapshot,
    uint256[] memory statePtrs,
    bytes32 agentRoot,
    uint32 notaryIndex,
    uint256 sigIndex
) internal returns (bytes memory attPayload) {
    // Attestation nonce is its index in `_attestations` array
    uint32 attNonce = uint32(_attestations.length);
    bytes32 snapGasHash = GasDataLib.snapGasHash(snapshot.snapGas());
    SummitAttestation memory summitAtt =
_toSummitAttestation(snapshot.calculateRoot(), agentRoot, snapGasHash);
    attPayload = _formatSummitAttestation(summitAtt, attNonce);
    _latestAttNonce[notaryIndex] = attNonce;
    /// @dev Add a single element to both `_attestations` and `_notarySnapshots`,
    /// enforcing the (_attestations.length == _notarySnapshots.length) invariant.
    _attestations.push(summitAtt);
    _notarySnapshots.push(SummitSnapshot(statePtrs, sigIndex));
    // Emit event with raw attestation data
    emit AttestationSaved(attPayload);
}
```

Figure 1.1: The `_saveNotarySnapshot` function in `SnapshotHub.sol`

The explicit casting to a lower `bytes` type truncates the higher-order bits. Therefore, the value of the `attNonce` variable will roll over to the `_attestations.length` attribute modulo 2^32 after adding 2^32 attestations. However, the snapshot submitted by the notary and the corresponding attestation will still be stored successfully. This will result in the following issues:

1. The `_latestAttNonce` variable for the notary will be set to a lower value than earlier.
2. The users and off-chain components fetching the attestation with the nonce in the attestation payload will see an incorrect attestation.
3. The users and off-chain components fetching the snapshot with the nonce in the attestation payload will see an incorrect snapshot.
4. The attestation payload containing the wrong attestation nonce will be stored in the `Destination` contract on the Synapse chain.

The `attestationPayload` returned by the `_saveNotarySnapshot` function is then forwarded to the `Destination` contract on the Synapse chain. The wrong attestation nonce stored in the `Destination` contract is then used in the `submitReceipt` function in the `Inbox` contract, as shown in figure 1.2:

```
function submitReceipt(
    bytes memory rcptPayload,
    bytes memory rcptSignature,
    uint256 paddedTips,
    bytes32 headerHash,
    bytes32 bodyHash
) external returns (bool wasAccepted) {
    // Struct to get around stack too deep error.
    ReceiptInfo memory info;
    // This will revert if payload is not a receipt
    Receipt rcpt = rcptPayload.castToReceipt();
    // This will revert if the receipt signer is not a known Notary
    (info.rcptNotaryStatus, info.notary) = _verifyReceipt(rcpt, rcptSignature);
    // Receipt Notary needs to be Active
    info.rcptNotaryStatus.verifyActive();
    info.attNonce =
IExecutionHub(destination).getAttestationNonce(rcpt.snapshotRoot());
    if (info.attNonce == 0) revert IncorrectSnapshotRoot();
    ...
    wasAccepted = InterfaceSummit(summit).acceptReceipt({
        rcptNotaryIndex: info.rcptNotaryStatus.index,
        attNotaryIndex: info.attNotaryStatus.index,
        sigIndex: sigIndex,
        attNonce: info.attNonce,
        paddedTips: paddedTips,
        rcptPayload: rcptPayload
    });
    if (wasAccepted) {
        emit ReceiptAccepted(info.rcptNotaryStatus.domain, info.notary, rcptPayload,
rcptSignature);
    }
}
```

Figure 1.2: The `submitReceipt` function in `Inbox.sol`

This leads to the wrong attestation nonce being submitted to the `Summit` contract as part of the receipt. The snapshot tips for the states included in the attestation will be distributed to the wrong guards and notaries—those who submitted the states of the older snapshot stored at the index of the wrong attestation nonce.

In particular, the attestation nonce value `0` will be used for the snapshot stored at the `2^32` index, and the receipt for this attestation cannot be submitted to the `Inbox` contract. This will result in non-payment of the tips for all the messages in this attestation.

Moreover, the off-chain components listening to the attestation events will need to be upgraded to handle these invalid attestation nonces even after fixing the smart contracts.

Exploit Scenario

The protocol is configured with 50 notaries active on 50 blockchain networks. Users are sending messages on origin chains at a rate of 2,000,000 messages per day (24 transactions per second * 86,400 seconds in a day). The notaries submit a snapshot for every 32 messages. This will lead to an overflow of the attestation nonce in approximately four years. This overflow goes unnoticed for some time, and smart contracts have multiple attestations and receipts with wrong nonce values.

Recommendations

Short term, consider making the following changes:

1. Analyze all the bridged network transaction throughputs to calculate an upper bound on the number of snapshots for a desired time frame. Use a higher-bit length `uint` type to accommodate the calculated upper bound.

2. Add a check in the `_saveNotarySnapshot` function to revert the attestation nonce overflow to prevent using the wrong nonce values. Consider using the `SafeCast` contract from the OpenZeppelin library to safely cast integer values.

Long term, analyze the upper bound on the values being cast down to ensure that they do not overflow. Add a comment in the code explaining the upper-bound analysis. Always check the casting overflows to avoid use of the overflowed value going unnoticed.

2. Initialization functions are vulnerable to front-running

Severity: **Informational**	Difficulty: **Low**
Type: Timing	Finding ID: TOB-PSC-02
Target: All the smart contracts	

Description

Several upgradable contracts have initialization functions that can be front-run, which would allow an attacker to incorrectly initialize the contracts.

The upgradable smart contracts using the `delegatecall` proxy pattern use an `initialize` function to set initial values of the state variables:

```
function initialize(bytes32 agentRoot) external initializer {
    // Initialize Ownable: msg.sender is set as "owner"
    __Ownable_init();
    // Initialize ReeentrancyGuard
    __ReentrancyGuard_init();
    // Set Agent Merkle Root in Light Manager
    if (localDomain != SYNAPSE_DOMAIN) {
        _nextAgentRoot = agentRoot;
        InterfaceLightManager(address(agentManager)).setAgentRoot(agentRoot);
        destStatus.agentRootTime = uint40(block.timestamp);
    }
    // No need to do anything on Synapse Chain, as the agent root is set in
BondingManager
}
```

Figure 2.1: The `initialize` function in `Destination.sol`

An attacker could front-run these functions and initialize the contracts with malicious values. The system relies on complex interactions between multiple smart contracts whose addresses are set as immutable state variables during deployment. As a result, any such attack that misconfigures just one part of the system would require the entire system or multiple parts of the system to be redeployed.

This issue is prevalent in the following contracts:

1. `Destination`
2. `Origin`
3. `GasOracle`

4. `Summit`
5. `BondingManager`
6. `LightManager`
7. `Inbox`
8. `LightInbox`

Exploit Scenario

Bob deploys the `BondingManager` contract. Eve front-runs the `BondingManager` initialization and sets her own address as the owner of the contract. She can now prevent the protocol team from adding new agents or potentially add arbitrary agents herself. The protocol team will need to either fully or partially redeploy the system.

Recommendations

Short term, either use a factory pattern that will prevent front-running of the initialization functions, or ensure that the deployment scripts have robust protections against front-running attacks.

Long term, carefully review the Solidity documentation, especially the Warning section, and the pitfalls of using the `delegatecall` proxy pattern.

3. Notaries can execute arbitrary messages to arbitrary receivers

Severity: **High**	Difficulty: **Low**
Type: Data Validation	Finding ID: TOB-PSC-03
Target: `inbox/LightInbox.sol`, `Destination.sol`	

Description

A notary can submit a malicious attestation with an `optimisticPeriod` parameter of zero and execute any base message within the same transaction. This allows notaries to execute arbitrary base messages without giving guards enough time to dispute the attestation.

Whenever a message is sent through the `sendBaseMessage` function of the `Origin` contract, guards relay this message to the `Summit` contract on the Synapse chain, after which a notary relays it to the `Destination` contract on the destination chain.

A guard first needs to complete the propagation to the `Summit` contract by calling the `submitSnapshot` function on the `Inbox` contract of the Synapse chain:

```
function submitSnapshot(bytes memory snapPayload, bytes memory snapSignature)
    external
    returns (bytes memory attPayload, bytes32 agentRoot_, uint256[] memory snapGas)
{
    // This will revert if payload is not a snapshot
    Snapshot snapshot = snapPayload.castToSnapshot();
    // This will revert if the signer is not a known Guard/Notary
    (AgentStatus memory status, address agent) =
        _verifySnapshot({snapshot: snapshot, snapSignature: snapSignature,
verifyNotary: false});
    // Check that Agent is active
    status.verifyActive();
    // Store Agent signature for the Snapshot
    uint256 sigIndex = _saveSignature(snapSignature);
    if (status.domain == 0) {
        // Guard that is in Dispute could still submit new snapshots, so we don't
check that
        InterfaceSummit(summit).acceptGuardSnapshot({
            guardIndex: status.index,
            sigIndex: sigIndex,
            snapPayload: snapPayload
        });
    } else {
    ...
```

Figure 3.1: Snippet of the `submitSnapshot` function in `Inbox.sol`

Then a notary calls the `submitSnapshot` function to submit a snapshot to the `Inbox` contract. This calls the `_saveNotarySnapshot` function and creates an attestation payload:

```
function _saveNotarySnapshot(
    Snapshot snapshot,
    uint256[] memory statePtrs,
    bytes32 agentRoot,
    uint32 notaryIndex,
    uint256 sigIndex
) internal returns (bytes memory attPayload) {
    // Attestation nonce is its index in `_attestations` array
    uint32 attNonce = uint32(_attestations.length);
    bytes32 snapGasHash = GasDataLib.snapGasHash(snapshot.snapGas());
    SummitAttestation memory summitAtt =
_toSummitAttestation(snapshot.calculateRoot(), agentRoot, snapGasHash);
    attPayload = _formatSummitAttestation(summitAtt, attNonce);
    _latestAttNonce[notaryIndex] = attNonce;
    /// @dev Add a single element to both `_attestations` and `_notarySnapshots`,
    /// enforcing the (_attestations.length == _notarySnapshots.length) invariant.
    _attestations.push(summitAtt);
    _notarySnapshots.push(SummitSnapshot(statePtrs, sigIndex));
    // Emit event with raw attestation data
    emit AttestationSaved(attPayload);
}
```

Figure 3.2: The `_saveNotarySnapshot` function in `SnapshotHub.sol`

The notary signs the attestation payload and submits it to the `submitAttestation` function of the `LightInbox` contract on the destination chain. The `submitAttestation` function verifies the notary signature and forwards the attestation payload to the `Destination` contract to be stored locally:

```
function submitAttestation(
    bytes memory attPayload,
    bytes memory attSignature,
    bytes32 agentRoot_,
    uint256[] memory snapGas_
) external returns (bool wasAccepted) {
    // This will revert if payload is not an attestation
    Attestation att = attPayload.castToAttestation();
    // This will revert if signer is not an known Notary
    (AgentStatus memory status, address notary) = _verifyAttestation(att,
attSignature);
    // Check that Notary is active
    status.verifyActive();
    // Check if Notary is active on this chain
    _verifyNotaryDomain(status.domain);

    ...
    // This will revert if Notary is in Dispute
    wasAccepted = InterfaceDestination(destination).acceptAttestation({
        notaryIndex: status.index,
        sigIndex: sigIndex,
        attPayload: attPayload,
        agentRoot: agentRoot_,
        snapGas: snapGas
    });
    if (wasAccepted) {
        emit AttestationAccepted(status.domain, notary, attPayload, attSignature);
    }
}
```

Figure 3.3: Snippet of the `submitAttestation` function in `LightInbox.sol`

Once this is done, anyone can execute the message by providing the `originProof` and the `snapProof` variables to the `Destination` contract's `execute` function:

```
function execute(
    bytes memory msgPayload,
    bytes32[] calldata originProof,
    bytes32[] calldata snapProof,
    uint256 stateIndex,
    uint64 gasLimit
) external nonReentrant {
    // This will revert if payload is not a formatted message payload
    Message message = msgPayload.castToMessage();
    Header header = message.header();
    bytes32 msgLeaf = message.leaf();
    // Ensure message was meant for this domain
    if (header.destination() != localDomain) revert IncorrectDestinationDomain();
    // Check that message has not been executed before
    ReceiptData memory rcptData = _receiptData[msgLeaf];
    if (rcptData.executor != address(0)) revert AlreadyExecuted();
    // Check proofs validity
    SnapRootData memory rootData = _proveAttestation(header, msgLeaf, originProof,
snapProof, stateIndex);
    // Check if optimistic period has passed
    uint256 proofMaturity = block.timestamp - rootData.submittedAt;
    if (proofMaturity < header.optimisticPeriod()) revert MessageOptimisticPeriod();
    uint256 paddedTips;
    bool success;
    // Only Base/Manager message flags exist
    if (header.flag() == MessageFlag.Base) {
        // This will revert if message body is not a formatted BaseMessage payload
        BaseMessage baseMessage = message.body().castToBaseMessage();
        success = _executeBaseMessage(header, proofMaturity, gasLimit, baseMessage);
        paddedTips = Tips.unwrap(baseMessage.tips());
    } else {
    ...
```

Figure 3.4: Snippet of the `execute` function in `ExecutionHub.sol`

The `execute` function forces a delay in the execution of the message from the time the attestation was submitted so that guards can dispute the notary upon submission of a malicious attestation. The value of the `optimisticPeriod`, provided by the user when sending a message from the origin chain, is used to enforce this execution delay.

However, a notary can completely skip all the previous required steps and submit an arbitrary attestation to the `LightInbox` contract because the contract cannot verify that the attestation is tied to a snapshot on the Synapse chain. A notary can submit an attestation for a base message with an `optimisticPeriod` of zero and execute the message in one transaction, not giving the guards a chance to dispute the notary.

Exploit Scenario

Eve, a notary active on a destination chain, crafts a malicious message to a target contract with the `optimisticPeriod` set to zero. The message is used to create a snapshot and attestation. Eve then submits the crafted attestation and executes the message in a single transaction, which allows her to execute the attack on the target contract without being disputed.

Recommendations

Short term, take the following actions:

1. Enforce a minimum value for the `optimisticPeriod` saved in the header of the message on both the origin chain and the destination chain.

2. Document the importance of validating the `optimisticPeriod` to educate dapp developers integrating with the Synapse Protocol.

Long term, analyze the protocol's security model to recognize potential malicious behavior and ensure that the mitigation does not depend on user inputs in the implementation. Always validate and restrict user inputs critical to the system's security properties.

4. Lack of upper bound on the summit tip

Severity: **Informational**	Difficulty: **High**
Type: Data Validation	Finding ID: TOB-PSC-04
Target: `GasOracle.sol`	

Description

The lack of an upper bound when setting the `summitTip` state variable can lead to an unintentionally high value being set, resulting in loss to users.

When a user submits a message to the system, they must pay a tip to ensure the verification and execution of their message. This tip serves as an incentive for guards and notaries, rewarding them with the native assets of the originating chain for their work in verifying and executing cross-chain messages.

The `setSummitTip` function allows the contract owner to specify a `summitTip` that will be included in the total tip calculations of all messages passed through the Synapse Protocol.

```
function setSummitTip(uint256 summitTipWei) external onlyOwner {
    _summitTipWei = summitTipWei;
}
```

Figure 4.1: The `setSummitTip` function in `GasOracle.sol`

However, there is no upper bound on the `summitTip` amount; therefore, it could be set to any value. An excessive tip amount (e.g., resulting from a typo) may not be noticed until it causes loss to users and results in disruptions.

Recommendations

Short term, set an upper bound for the `summitTip` state variable. The upper bound should be high enough to encompass any reasonable value but low enough to reject mistakenly or maliciously entered values that would result in unsustainably high fees.

Long term, carefully document the caller-specified values that dictate the financial properties of the system and ensure that they are properly constrained.

5. The bridge may not function for a destination chain

Severity: **High**	Difficulty: **High**
Type: Configuration	Finding ID: TOB-PSC-05
Target: `manager/LightManager.sol`, `inbox/LightInbox.sol`	

Description

Configuring the `Destination` contract with an incorrect or zero `agentRoot` argument will prevent notaries from registering on that destination chain, permanently halting the Synapse Protocol on that chain.

The protocol assigns notaries to relay messages from one chain to another to make them available to be executed on the destination chain. To become a notary, an account must be added as a notary agent on the Synapse chain. This is done by the owner of the `BondingManager` contract using the `addAgent` function:

```
function addAgent(uint32 domain, address agent, bytes32[] memory proof) external
onlyOwner {
    if (domain == SYNAPSE_DOMAIN) revert SynapseDomainForbidden();
    AgentStatus memory status = _storedAgentStatus(agent);
    uint32 index;
    bytes32 oldValue;
    if (status.flag == AgentFlag.Unknown) {
        if (_agents.length >= type(uint32).max) revert MerkleTreeFull();
        index = uint32(_agents.length);
        _agents.push(agent);
        _domainAgents[domain].push(agent);
    } else if (status.flag == AgentFlag.Resting && status.domain == domain) {
        index = status.index;
        oldValue = _agentLeaf(AgentFlag.Resting, domain, agent);
    } else {
        revert AgentCantBeAdded();
    }
    // This will revert if the proof for the old value is incorrect
    _updateLeaf(oldValue, proof, AgentStatus(AgentFlag.Active, domain, index),
agent);
}
```

Figure 5.1: The `addAgent` function in `BondingManager.sol`

All guards and notaries are represented as leaves of a Merkle tree. The root of this Merkle tree is used to verify that an agent was added to the Synapse chain before being allowed to register on another chain. The root is provided as an argument during the initialization of the `Destination` contract:

```
function initialize(bytes32 agentRoot) external initializer {
    // Initialize Ownable: msg.sender is set as "owner"
    __Ownable_init();
    // Initialize ReeentrancyGuard
    __ReentrancyGuard_init();
    // Set Agent Merkle Root in Light Manager
    if (localDomain != SYNAPSE_DOMAIN) {
        _nextAgentRoot = agentRoot;
        InterfaceLightManager(address(agentManager)).setAgentRoot(agentRoot);
        destStatus.agentRootTime = uint40(block.timestamp);
    }
    // No need to do anything on Synapse Chain, as the agent root is set in
BondingManager
}
```

Figure 5.2: The `initialize` function in `Destination.sol`

If an account is a part of the agent Merkle tree, it can use the `updateAgentStatus` function of the `LightManager` contract to register on the destination chain. The function will verify that the agent is a part of the Merkle tree and then update their status in the `LightManager` contract:

```
function updateAgentStatus(address agent, AgentStatus memory status, bytes32[]
memory proof) external {
    address storedAgent = _agents[status.index];
    if (storedAgent != address(0) && storedAgent != agent) revert
IncorrectAgentIndex();
    // Reconstruct the agent leaf: flag should be Active
    bytes32 leaf = _agentLeaf(status.flag, status.domain, agent);
    bytes32 root = agentRoot;
    // Check that proof matches the latest merkle root
    if (MerkleMath.proofRoot(status.index, leaf, proof, AGENT_TREE_HEIGHT) != root)
revert IncorrectAgentProof();
    // Save index => agent in the map
    if (storedAgent == address(0)) {
        _agents[status.index] = agent;
        _agentIndexes[agent] = status.index;
    }
    // Update the agent status against this root
    _agentMap[root][agent] = status;
    emit StatusUpdated(status.flag, status.domain, agent);
    // Notify local AgentSecured contracts, if agent flag is Slashed
    if (status.flag == AgentFlag.Slashed) {
        // This will revert if the agent has been slashed earlier
        _resolveDispute(status.index, msg.sender);
    }
}
```

Figure 5.3: The `updateAgentStatus` function in `LightManager.sol`

Once a notary is registered, they can provide attestations to verify the cross-chain messages and to forward new agentRoots from the Synapse chain to the destination chain so new agents can register on the destination chain.

```
function submitAttestation(
    bytes memory attPayload,
    bytes memory attSignature,
    bytes32 agentRoot_,
    uint256[] memory snapGas_
) external returns (bool wasAccepted) {
    // Code to verify the attestation …

    wasAccepted = InterfaceDestination(destination).acceptAttestation({
        notaryIndex: status.index,
        sigIndex: sigIndex,
        attPayload: attPayload,
        agentRoot: agentRoot_,
        snapGas: snapGas
    });
    if (wasAccepted) {
        emit AttestationAccepted(status.domain, notary, attPayload, attSignature);
    }
}
```

Figure 5.4: Snippet of the `submitAttestation` function in `LightInbox.sol`

However, if the `Destination` contract is misconfigured such that an incorrect or zero `agentRoot` is provided, no notary can register on that destination chain, permanently halting the Synapse Protocol on that chain.

Additionally, a new `agentRoot` can be provided only by currently active notaries, along with their attestations. As a result, if all the notaries are slashed or inactive, the destination chain will be permanently halted due to no new notaries being able to register.

Exploit Scenario

The Synapse Protocol team decides to deploy the system on a new chain but accidentally initializes the `agentRoot` on the `Destination` contract to an invalid value. Since no agents can register, no attestations can be made, permanently halting the functionality of the Synapse Protocol on this chain.

Recommendations

Short term, add an admin function (with the `onlyOwner` modifier) to update the `agentRoot` in the `LightManager` contract. Consider the following approaches to restrict the owner from maliciously or accidentally updating the `agentRoot`:

1. Allow the owner to update the `agentRoot` only if it has not been updated for a certain period.

2. Implement this feature as a two-step function with a time delay. Use the time delay to detect suspicious activity and change ownership.

Long term, set up local fork testing in the test suite for contract upgrades and new deployments to ensure the contract state is correctly initialized. Consider defining network uptime criteria (e.g., the number of active notaries) and setting up a monitoring system and response plan for when the network is experiencing congestion or is at risk of halting.

6. Guards can grieve honest notaries by disputing them during peak traffic

Severity: **High**	Difficulty: **Low**
Type: Denial of Service	Finding ID: TOB-PSC-06
Target: `inbox/LightInbox.sol`, `manager/AgentManager.sol`	

Description

Guards can dispute honest notaries on their domain to stop them from earning tips during peak traffic towards their domain chain.

The protocol allows guards to dispute a notary to stop them from executing a malicious message on the destination chain. However, the dispute resolution happens on the origin chain or the Synapse chain, depending on the dispute payload. The propagation of the new `agentRoot`, with the slashed agent, to the destination chain takes a minimum of one day because of the delay in the `agentRoot` passing mechanism:

```
function passAgentRoot() public returns (bool rootPassed, bool rootPending) {
    // Agent root is not passed on Synapse Chain, as it could be accessed via
BondingManager
    if (localDomain == SYNAPSE_DOMAIN) return (false, false);
    bytes32 oldRoot = IAgentManager(agentManager).agentRoot();
    bytes32 newRoot = _nextAgentRoot;
    // Check if agent root differs from the current one in LightManager
    if (oldRoot == newRoot) return (false, false);
    DestinationStatus memory status = destStatus;
    // Invariant: Notary who supplied `newRoot` was registered as active against
`oldRoot`
    // So we just need to check the Dispute status of the Notary
    if (_isInDispute(status.notaryIndex)) {
        // Remove the pending agent merkle root, as its signer is in dispute
        _nextAgentRoot = oldRoot;
        return (false, false);
    }
    // Check if agent root optimistic period is over
    if (status.agentRootTime + AGENT_ROOT_OPTIMISTIC_PERIOD > block.timestamp) {
        // We didn't pass anything, but there is a pending root
        return (false, true);
    }
    // `newRoot` signer was not disputed, and the root optimistic period is over.
    // Finally, pass the Agent Merkle Root to LightManager
    InterfaceLightManager(address(agentManager)).setAgentRoot(newRoot);
    return (true, false);
}
```

Figure 6.1: Snippet of the `passAgentRoot` function in `Destination.sol`

The protocol allows the owner of the `LightManager` contract on the destination chain to resolve a stuck dispute, but this function also enforces a delay of four hours from the time of the last attestation submission. During peak traffic, other notaries will keep submitting attestations, and the owner will be unable to resolve the dispute:

```
function resolveStuckDispute(uint32 domain, address slashedAgent) external onlyOwner
{
    AgentDispute memory slashedDispute = _agentDispute[_getIndex(slashedAgent)];
    if (slashedDispute.flag == DisputeFlag.None) revert DisputeNotOpened();
    if (slashedDispute.flag == DisputeFlag.Slashed) revert DisputeAlreadyResolved();
    // Check if there has been no fresh data from the Notaries for a while.
    (uint40 snapRootTime,,) = InterfaceDestination(destination).destStatus();
    if (block.timestamp < FRESH_DATA_TIMEOUT + snapRootTime) revert
DisputeNotStuck();
    // This will revert if domain doesn't match the agent's domain.
    _slashAgent({domain: domain, agent: slashedAgent, prover: address(0)});
}
```

Figure 6.2: The `resolveStuckDispute` function in `AgentManager.sol`

The attestations submitted by a disputed notary are rejected by the `Destination` contract. Therefore, a disputed notary cannot earn tips by submitting attestations for their domain.

We classify this issue as having a low difficulty because there is no cost to the guards for a malicious dispute. However, we acknowledge that the The Organisation team plans to implement the staking feature in the future, which will make this vulnerability costly to exploit.

Exploit Scenario

The protocol assigns a guard, Eve, and two notaries, Alice and Bob, to relay messages to the Polygon chain. Alice colludes with Eve to earn all the tips from an upcoming airdrop on the Polygon chain. Eve disputes Bob on the Polygon chain just before the airdrop launch time. Bob loses this opportunity to make a profit from the airdrop traffic, and Alice makes a good profit by relaying and executing all the messages.

Recommendations

Short term, analyze the cost of disputes to notaries and enforce appropriate punishment to discourage malicious disputes from guards.

Long term, minimize trust assumptions for system actors. Design incentives for honest behavior and punishments for malicious behavior after analyzing the potential loss to the victims of an attack. Discouraging malicious behavior is as important as encouraging honest behavior for proper incentive alignment.

7. A portion of the message value gets locked in the Origin contract	
Severity: **Medium**	Difficulty: **Low**
Type: Data Validation	Finding ID: TOB-PSC-07
Target: `Origin.sol, libs/stack/Tips.sol`	

Description

The rounding down of the `delta` value of minimum tips and the value sent by a user results in a portion of the excess amount becoming locked in the `Origin` contract.

The `sendBaseMessage` function in the `Origin` contract computes the minimum tips required to successfully relay the message to the destination chain. It then matches the amount of the tips with the `msg.value` variable and adds the excess amount to the delivery tip:

```
function matchValue(Tips tips, uint256 newValue) internal pure returns (Tips
newTips) {
    uint256 oldValue = tips.value();
    if (newValue < oldValue) revert TipsValueTooLow();
    // We want to increase the delivery tip, while keeping the other tips the same
    unchecked {
        uint256 delta = (newValue - oldValue) >> TIPS_GRANULARITY;
        // `delta` fits into uint224, as TIPS_GRANULARITY is 32, so this never
overflows uint256.
        // In practice, this will never overflow uint64 as well, but we still check
it just in case.
        if (delta + tips.deliveryTip() > type(uint64).max) revert TipsOverflow();
        // Delivery tips occupy lowest 8 bytes, so we can just add delta to the tips
value
        // to effectively increase the delivery tip (knowing that delta fits into
uint64).
        newTips = Tips.wrap(Tips.unwrap(tips) + delta);
    }
}
```

Figure 7.1: The `matchValue` function in `Tips.sol`

As shown in figure 7.1, the `delta` value, which is the difference between the user-provided value and the computed tips value, is shifted right by 32 bits. This truncates the least significant 32 bits from the excess amount, and the truncated value is added to the tips. However, if these least significant 32 bits are nonzero, then the amount corresponding to the bits will be stuck in the `Origin` contract because the difference between the final tips value and the `msg.value` is not returned to the caller.

Exploit Scenario

The minimum tip required to send the ping message to the Polygon chain from the Ethereum chain is 9*(2^32). Alice calls the `sendBaseMessage` function to send this ping message with a value of 38754705664 wei. After the execution of this transaction, the 100000000 wei of ETH are stuck in the `Origin` contract permanently.

Recommendations

Short term, to ensure accumulated excess tips are not locked in the `Origin` contract, consider the following changes:

1. Add a new function in the `Origin` contract to send base messages. This function should compute the difference between the final tips value and the `msg.value` and return the excess amount to the user.

2. Add an owner function to withdraw excess tip amounts from the `Origin` contract.

Since the maximum loss of value can be 2^32 wei and returning this value to the user will add a gas cost to the new function for sending base messages, users can decide which function to call depending on the gas price of the origin blockchain.

Long term, document the expected rounding direction for every arithmetic operation and follow it to ensure that rounding does not lead to loss for users or the protocol. Use Echidna to identify issues arising from the loss of precision due to rounding down.

8. Resolving stuck disputes can be indefinitely delayed

Severity: **Medium**	Difficulty: **High**
Type: Data Validation	Finding ID: TOB-PSC-08
Target: `manager/AgentManager.sol`	

Description

Notaries can prevent the contract owner from resolving stuck disputes by providing new attestations at regular intervals, resulting in potential loss to the disputed notaries, or guards being unable to dispute the malicious activity of the active notaries.

The Synapse Protocol allows the owner to resolve disputes that have not been resolved for at least four hours (the value of the `FRESH_DATA_TIMEOUT` constant) after the last attestation has been submitted. The owner can resolve disputes using the `resolveStuckDispute` function in the `AgentManager` contract, which is inherited by the `BondingManager` and `LightManager` contracts:

```
function resolveStuckDispute(uint32 domain, address slashedAgent) external onlyOwner
{
    AgentDispute memory slashedDispute = _agentDispute[_getIndex(slashedAgent)];
    if (slashedDispute.flag == DisputeFlag.None) revert DisputeNotOpened();
    if (slashedDispute.flag == DisputeFlag.Slashed) revert DisputeAlreadyResolved();
    // Check if there has been no fresh data from the Notaries for a while.
    (uint40 snapRootTime,,) = InterfaceDestination(destination).destStatus();
    if (block.timestamp < FRESH_DATA_TIMEOUT + snapRootTime) revert
DisputeNotStuck();
    // This will revert if domain doesn't match the agent's domain.
    _slashAgent({domain: domain, agent: slashedAgent, prover: address(0)});
}
```

Figure 8.1: The `resolveStuckDispute` function in `AgentManager.sol`

The function slashes the provided agent only if at least the number of seconds in `FRESH_DATA_TIMEOUT` have passed since the last attestation submission. The last attestation submission time is stored in the destination contract as the `destStatus.snapRootTime` attribute. It is updated in the `_saveAgentRoot` function, which is called upon submission of every attestation:

```
function _saveAgentRoot(bool rootPending, bytes32 agentRoot, uint32 notaryIndex)
    internal
    returns (DestinationStatus memory status)
{
    status = destStatus;
    // Update the timestamp for the latest snapshot root
    status.snapRootTime = uint40(block.timestamp);
    // No need to save agent roots on Synapse Chain, as they could be accessed via
BondingManager
    // Don't update agent root, if there is already a pending one
    // Update the data for latest agent root only if it differs from the saved one
    if (localDomain != SYNAPSE_DOMAIN && !rootPending && _nextAgentRoot !=
agentRoot) {
        status.agentRootTime = uint40(block.timestamp);
        status.notaryIndex = notaryIndex;
        _nextAgentRoot = agentRoot;
        emit AgentRootAccepted(agentRoot);
    }
}
```

Figure 8.2: The `_saveAgentRoot` function in `Destination.sol`

Competing notaries can submit attestations at regular intervals to update the `snapRootTime` attribute to make the `resolveStuckDispute` function revert, potentially permanently locking a guard and notary in a dispute. Because of this, the disputed guard cannot dispute other malicious notaries, and the disputed notary cannot submit attestations.

Exploit Scenario

The protocol assigns a guard, Eve, and two notaries, Alice and Bob, to relay messages to the Polygon chain. To increase his earnings, Bob colludes with Eve to create a false dispute with Alice, which can be resolved only by the contract owner. Bob proceeds to submit attestations at regular intervals so that the owner is unable to resolve the dispute. Alice is now permanently locked in a dispute and unable to provide attestations.

Recommendations

Short term, consider the following changes:

1. Add a function to allow the owner to initiate the resolution of a dispute. Save the timestamp of the resolution initiation.

2. Add another function to complete the resolution of a dispute while delaying the completion by checking the current time against the resolution initiation time.

3. Use this time delay to monitor any malicious activity from the owner and transfer ownership if required.

Long term, evaluate all time-delay features to ensure that the start time and end time do not rely on values that can be controlled or influenced by other users.

9. Risk of message execution reordering

Severity: **Medium**	Difficulty: **Low**
Type: Timing	Finding ID: TOB-PSC-09
Target: `Destination.sol`	

Description

A malicious user can execute the bridged messages in a specific order to extract profit, potentially causing a loss to the users that initiated the messages.

The protocol allows multiple messages to be submitted in a single snapshot and attestation to prepare them for execution. Once a guard and notary have submitted the snapshot and attestation, anyone can execute the messages by calling the `execute` function of the `Destination` contract:

```
function execute(
    bytes memory msgPayload,
    bytes32[] calldata originProof,
    bytes32[] calldata snapProof,
    uint256 stateIndex,
    uint64 gasLimit
) external nonReentrant {
    // This will revert if payload is not a formatted message payload
    Message message = msgPayload.castToMessage();
    Header header = message.header();
    bytes32 msgLeaf = message.leaf();
    // Ensure message was meant for this domain
    if (header.destination() != localDomain) revert IncorrectDestinationDomain();
    // Check that message has not been executed before
    ReceiptData memory rcptData = _receiptData[msgLeaf];
    if (rcptData.executor != address(0)) revert AlreadyExecuted();
    // Check proofs validity
    SnapRootData memory rootData = _proveAttestation(header, msgLeaf, originProof,
snapProof, stateIndex);
    // Check if optimistic period has passed
    uint256 proofMaturity = block.timestamp - rootData.submittedAt;
    if (proofMaturity < header.optimisticPeriod()) revert MessageOptimisticPeriod();
    uint256 paddedTips;
    bool success;
    // Only Base/Manager message flags exist
    if (header.flag() == MessageFlag.Base) {
        // This will revert if message body is not a formatted BaseMessage payload
        BaseMessage baseMessage = message.body().castToBaseMessage();
        success = _executeBaseMessage(header, proofMaturity, gasLimit, baseMessage);
        paddedTips = Tips.unwrap(baseMessage.tips());
    } else {
        // gasLimit is ignored when executing manager messages
        success = _executeManagerMessage(header, proofMaturity, message.body());
    }
    ...
    }
}
```

Figure 9.1: A snippet of the `execute` function in `ExecutionHub.sol`

The contract also implements the `multicall` function, which allows multiple calls to be bundled together and executed in a single transaction:

```
function multicall(Call[] calldata calls) external returns (Result[] memory
callResults) {
    uint256 amount = calls.length;
    callResults = new Result[](amount);
    Call calldata call_;
    for (uint256 i = 0; i < amount;) {
        call_ = calls[i];
        Result memory result = callResults[i];
        // We perform a delegate call to ourselves here. Delegate call does not
modify `msg.sender`, so
        // this will have the same effect as if `msg.sender` performed all the calls
themselves one by one.
        // solhint-disable-next-line avoid-low-level-calls
        (result.success, result.returnData) =
address(this).delegatecall(call_.callData);
        // solhint-disable-next-line no-inline-assembly
        assembly {
            // Revert if the call fails and failure is not allowed
            // `allowFailure := calldataload(call_)` and `success := mload(result)`
            if iszero(or(calldataload(call_), mload(result))) {
                // @audit check if this is correct
                // Revert with `0x4d6a2328` (function selector for
`MulticallFailed()`)
                mstore(0x00,
0x4d6a232800000000000000000000000000000000000000000000000000000000)
                revert(0x00, 0x04)
            }
        }
        unchecked {
            ++i;
        }
    }
}
```

Figure 9.2: The `multicall` function in `MultiCallable.sol`

Due to the `execute` function not enforcing a specific order of execution, a malicious user can bundle and reorder the execution of bridged messages to extract profit.

Exploit Scenario
A decentralized exchange integrates its protocol with the Synapse Protocol to allow users to make cross-chain swaps. Alice submits a message to swap from ETH to LINK in one of the DEX pools. Eve notices this and submits two of her own messages:

1. The first message swaps from ETH to LINK.
2. The second message swaps from LINK to ETH.

Eve then orders the bridged messages so that her first message is executed first, then Alice's message is executed, and finally, Eve's second message is executed. She calls the `multicall` function to easily do this in one transaction. This sequence benefits Eve through the increased price of the LINK token after Alice's swap.

Recommendations
Short term, highlight message execution reordering risks in the user- and developer-facing documentation. Encourage application developers to mitigate this risk when integrating with the protocol.

Long term, carefully consider the unpredictable nature of blockchain transactions, relays, and users executing transactions, and design the contracts to not depend on the transaction ordering.

10. Tip calculation can round down to zero

Severity: **Informational**	Difficulty: **High**
Type: Data Validation	Finding ID: TOB-PSC-10
Target: `libs/stack/Tips.sol`, `GasOracle.sol`	

Description

The tip calculation can round down to zero for chains with native assets having fewer than 10 decimals of precision, potentially halting the Synapse Protocol for the origin chain and allowing users to spam the protocol at no cost.

Whenever a user sends a cross-chain message by calling the `sendBaseMessage` function of the `Origin` contract on the origin chain, they must provide tips in the form of some amount of native assets. These tips are used to incentivize the guards and notaries to relay the message to the destination chain and incentivize other users to execute the provided message on the destination chain:

```
function sendBaseMessage(
    uint32 destination,
    bytes32 recipient,
    uint32 optimisticPeriod,
    uint256 paddedRequest,
    bytes memory content
) external payable returns (uint32 messageNonce, bytes32 messageHash) {
    // Check that content is not too large
    if (content.length > MAX_CONTENT_BYTES) revert ContentLengthTooBig();
    // This will revert if msg.value is lower than value of minimum tips
    Tips tips = _getMinimumTips(destination, paddedRequest,
content.length).matchValue(msg.value);
    Request request = RequestLib.wrapPadded(paddedRequest);
    // Format the BaseMessage body
    bytes memory body = BaseMessageLib.formatBaseMessage({
        sender_: msg.sender.addressToBytes32(),
        recipient_: recipient,
        tips_: tips,
        request_: request,
        content_: content
    });
    // Send the message
    return _sendMessage(destination, optimisticPeriod, MessageFlag.Base, body);
}
```

Figure 10.1: The sendBaseMessage function in `Origin.sol`

The minimum amount of tips is calculated using the `_getMinimumTips` function, which makes an external call to the `getMinimumTips` function of the `GasOracle` contract, as shown in figure 10.2:

```
function _getMinimumTips(uint32 destination, uint256 paddedRequest, uint256
contentLength)
    internal
    view
    returns (Tips)
{
    return
        TipsLib.wrapPadded(InterfaceGasOracle(gasOracle).getMinimumTips(destination,
paddedRequest, contentLength));
}
```

Figure 10.2: The `_getMinimumtips` function in `Origin.sol`

The `GasOracle` contract uses the latest previously saved gas data to determine the `summitTip`, `attestationTip`, `executionTip`, and `deliveryTip` variables, which are denominated in the wei unit. Then the contract encodes the individual tip values into a compressed number, truncating the least significant 32 bits:

```
function getMinimumTips(uint32 destination_, uint256 paddedRequest, uint256
contentLength)
    external
    view
    returns (uint256 paddedTips)
{
    ...

    // Use calculated values to encode the tips.
    return Tips.unwrap(
        TipsLib.encodeTips256({
            summitTip_: summitTip,
            attestationTip_: attestationTip,
            executionTip_: executionTip,
            deliveryTip_: deliveryTip
        })
    );
}
```

Figure 10.3: A snippet of the `getMinimumTips` function in `GasOracle.sol`

```
function encodeTips256(uint256 summitTip_, uint256 attestationTip_, uint256
executionTip_, uint256 deliveryTip_)
    internal
    pure
    returns (Tips)
{
    return encodeTips({
        summitTip_: uint64(summitTip_ >> TIPS_GRANULARITY),
        attestationTip_: uint64(attestationTip_ >> TIPS_GRANULARITY),
        executionTip_: uint64(executionTip_ >> TIPS_GRANULARITY),
        deliveryTip_: uint64(deliveryTip_ >> TIPS_GRANULARITY)
    });
}
```

Figure 10.4: A snippet of the `encodeTips256` function in `Tips.sol`

However, if the chain's native asset has fewer than 10 decimal places, all the significant bits will be discarded, resulting in the numbers being rounded down to zero. This can lead to notaries and guards not relaying the messages due to the absence of an incentive, which would lead to a complete halt of the protocol for the origin chain. The zero-tip value also allows users to send messages at no cost, which can be used to spam the Synapse Protocol.

Exploit Scenario

The team deploys the contracts on a new chain whose native asset has nine decimals. All the tip calculations round down to zero, halting the execution of messages on that chain until a contract upgrade is performed.

Recommendations

Short term, set up local-fork testing in the test suite to test that contracts behave as expected with all existing and upcoming target blockchain networks.

Long term, create a list of assumptions about the behavior of a blockchain that the system relies on (e.g., native asset decimals, available precompile contracts, gas costs, etc.). When deploying the system on a new blockchain, check the behavior against this list to ensure the system functions as expected.

11. Agents can collude to execute malicious manager messages

Severity: **High**	Difficulty: **High**
Type: Data Validation	Finding ID: TOB-PSC-11
Target: `Destination.sol`, `hubs/ExecutionHub.sol`	

Description

A notary can collude with a guard to avoid being disputed on an invalid message and thereby execute malicious manager messages. This allows the notary to steal tips from the `Origin` contract and slash any agent on the Synapse chain.

The protocol allows guards to dispute notaries when they detect an invalid message attestation submission. However, a guard can dispute only one notary at a time, and similarly, a notary can be disputed by only one guard at a time. This means that if a notary submits multiple invalid attestations, the notary will be disputed for the first submission, and messages in any of their other attestations cannot be executed while the notary is in dispute.

The protocol enforces an optimistic period delay from the submission of the attestation to the execution of the message to allow guards enough time to detect invalid attestations and dispute the notary before any malicious message can be executed.

However, when the dispute is resolved by slashing the guard, the notary becomes active, and messages included in their invalid attestations submitted before they were disputed can now be executed.

```
uint32 constant AGENT_ROOT_OPTIMISTIC_PERIOD = 1 days;
uint32 constant BONDING_OPTIMISTIC_PERIOD = 1 days;
```

Figure 11.1: Some constants defined in `Constants.sol`

The dispute resolution happens on the origin chain of the message and takes at least a day to update the agent tree root on the destination chain because of an enforced delay. This means that any message with an optimistic period delay of less than or equal to one day can be executed immediately after the dispute resolution if the dispute is resolved in favor of the notary.

In particular, the manager messages have an optimistic period delay of one day, and a manager message with arbitrary arguments can be executed by notaries by exploiting the flaw that the notary can be disputed only once. Notaries can be disputed for a valid attestation and become active again after the dispute resolution to execute the following attacks:

1. Stealing tips from the `Origin` contract by executing the `remoteWithdrawTips` function in the `LightManager` contract
2. Slashing any agent on the Synapse chain by executing the `remoteSlashAgent` function in the `BondingManager` contract

Exploit Scenario

Eve, a notary, colludes with Bob, a guard, to execute a manager message to steal tips from the `Origin` contract on any blockchain:

1. Eve submits a valid attestation and an invalid attestation and is disputed with Bob's signature on the valid attestation in a single transaction on the Polygon chain.
2. Now, even if other guards notice the invalid attestation, they cannot dispute Eve for it because she is already in dispute.
3. Eve then proceeds to slash Bob on the Synapse chain and waits for the slashed agent root to be propagated to the Polygon chain.
4. As soon the new agent root is updated in the `LightManager` contract on the Polygon chain, Eve executes a transaction to do the following two things:
    a. Update the status of Bob to slashed, which resolves the dispute for Eve and makes Eve active again
    b. Execute the malicious transaction submitted in the invalid attestation

Recommendations

Short term, make the following changes:

1. Enforce an optimistic period delay in the execution of manager messages that is more than the value of the `AGENT_ROOT_OPTIMISTIC_PERIOD` constant to ensure malicious messages cannot be executed.
2. Assign a new status to the agents becoming active after dispute resolution. Use this status to enforce a delay in the validation of their old submissions.
3. Implement a feature in the guard off-chain component to report invalid attestations after a notary becomes active following dispute resolution.

Long term, document valid system states and the state transitions allowed from each state. Ensure proper data validation and invalidation happen after each state transition.

12. Lack of a two-step process for ownership transfer

Severity: **High**	Difficulty: **High**
Type: Data Validation	Finding ID: TOB-PSC-12
Target: `base/MessagingBase.sol`	

Description

The `transferOwnership` function immediately transfers the ownership of the contract. The use of a single step to make such a critical change is error-prone; if the function is called with erroneous input, access to all owner-only functions will be lost permanently.

```
function transferOwnership(address newOwner) public virtual onlyOwner {
    require(newOwner != address(0), "Ownable: new owner is the zero address");
    _transferOwnership(newOwner);
}

function _transferOwnership(address newOwner) internal virtual {
    address oldOwner = _owner;
    _owner = newOwner;
    emit OwnershipTransferred(oldOwner, newOwner);
}
```

Figure 12.1: The `transferOwnership` function in `OwnableUpgradeable.sol`

Exploit Scenario

Alice invokes the `transferOwnership` function to change the contract owner of the `Summit` contract but accidentally enters the wrong address. She permanently loses access to the contract and is unable to add new agents or update the status of existing ones.

Recommendations

Short term, implement a two-step process for all irrecoverable critical operations. For ownership transfer, use OpenZeppelin's `Ownable2StepUpgradeable` instead of `OwnableUpgradeable` functions to mitigate this issue.

Long term, identify and document all possible actions that can be taken by privileged accounts, along with their associated risks. This will facilitate reviews of the codebase and prevent future mistakes.

13. Slashing mechanism makes guards and notaries likely targets of phishing campaigns	
Severity: **High**	Difficulty: **High**
Type: Data Validation	Finding ID: TOB-PSC-13
Target: All the smart contracts	

Description

An attacker can target guards and notaries in a phishing campaign to obtain their signature on an invalid payload and then use this signature to slash them for a reward.

The Synapse Protocol assigns agents to relay messages across the blockchain networks. A staking and slashing mechanism is used to discourage malicious behavior by these agents. The fraud prover is awarded part of the slashed amount to incentivize third parties to monitor the messages relayed by the agents.

All the functions accepting snapshots, attestations, receipts, and fraud proofs take an agent's signature on the message hash to verify that the message is submitted by an active agent. The protocol allows anyone to submit messages with a signature from an active agent. This makes the agents a target of phishing campaigns.

An attacker can launch a phishing campaign to make agents sign the hash of an invalid message and then use this signature to slash the agent for the fraud prover reward.

Exploit Scenario

Eve bonds a small stake and becomes a guard, while Alice bonds a larger stake and becomes a notary in the protocol. Eve creates a malicious airdrop called `ProofOfSignature` to claim the tokens, but a participant of the airdrop must sign a hash. Eve generates the hash of an invalid state, whose nonce is larger than the next valid nonce of the system, and asks Alice to sign the hash to obtain the free tokens. Alice signs the hash, and Eve uses this hash to submit a fake fraud report and slash Alice, earning a portion of her stake.

Recommendations

It is not possible to prevent phishing campaigns, but their likelihood of success can be reduced through user education and on-chain and off-chain mitigations.

Short term, consider using the EIP-712 encoding schema, which will help reduce the chance of agents signing malicious payloads. Additionally, document best practices for agents, including the following:

- Verify the message and its hash before signing it.
- Sign messages only if they are coming from a trusted source.
- Make hashing schemas public and provide tools to easily verify a message hash.
- Consider using a unique account for signing snapshots, attestations, and receipts. Ensure that this account is not used for anything else.

Long term, perform periodic security checks for agents and their off-chain components to keep them up to date with security best practices.

14. Tips for messages with an EOA recipient cannot be claimed

Severity: **Medium**	Difficulty: **Medium**
Type: Data Validation	Finding ID: TOB-PSC-14
Target: `hubs/ExecutionHub.sol`	

Description

The `execute` function reverts for messages with an externally owned account (EOA) recipient, which results in a loss to the agents relaying the message and the executors trying to execute the message.

The `Destination` contract inherits logic from the `ExecutionHub` contract, which allows any user to execute valid pending messages using the `execute` function. When executing base messages, this function calls an internal function, `_executeBaseMessage`, which uses a try-catch block to ensure that even if the message execution fails, a receipt will still be generated. This receipt can be used to claim the tips for relaying the message and attempting an execution.

```
function _executeBaseMessage(Header header, uint256 proofMaturity, uint64 gasLimit,
BaseMessage baseMessage)
    internal
    returns (bool)
{
    Request request = baseMessage.request();
    if (gasLimit < request.gasLimit()) revert GasLimitTooLow();
    address recipient = baseMessage.recipient().bytes32ToAddress();
    // Forward message content to the recipient, and limit the amount of forwarded
gas
    if (gasleft() <= gasLimit) revert GasSuppliedTooLow();
    try IMessageRecipient(recipient).receiveBaseMessage{gas: gasLimit}({
        origin: header.origin(),
        nonce: header.nonce(),
        sender: baseMessage.sender(),
        proofMaturity: proofMaturity,
        version: request.version(),
        content: baseMessage.content().clone()
    }) {
        return true;
    } catch {
        return false;
    }
}
```

Figure 14.1: The `_executeBaseMessage` function in `ExecutionHub.sol`

However, if the recipient of the message is not a contract, the call to the `_executeBaseMessage` function will revert without returning `true` or `false` and will not generate a receipt. This is because the compiler adds a contract existence check right before the try-catch block, which reverts the transaction if the receiver is not a contract.

Due to this issue, the executor and the agents that verified the message will incur a loss because they cannot withdraw their tips, and the message will be permanently stuck in the system.

Exploit Scenario
Eve sends a base message from Polygon to Ethereum but mistypes the receiver address. After the message is made available for execution by submitting snapshots and attestations, Alice attempts to execute the message on Ethereum. Due to the call reverting, Alice incurs a loss and is unable to withdraw her tips.

Recommendations
Short term, add a contract existence check right before the try-catch block in the `execute` function of the `ExecutionHub` contract to generate a receipt and return `false` if the recipient is not a contract.

Long term, define a list of system and function-level invariants and use smart contract fuzzing with Echidna to uncover these types of issues. Review the expected behavior of the try-catch statement described in appendix E and ensure the system properly handles each case.

15. Notaries can submit a high gas value to stop users from sending messages to a destination chain

Severity: **Medium**	Difficulty: **High**
Type: Denial of Service	Finding ID: TOB-PSC-15
Target: `Destination.sol`, `GasOracle.sol`	

Description

The `GasOracle` contract sources the `gasData` variable for remote blockchains from the bridged messages submitted by notaries. Notaries can submit a malicious attestation with a high gas value, which will discourage users from sending messages to the destination chain because of the high amount of tips required.

The Synapse Protocol uses the `GasOracle` contract to track gas prices for all the blockchains and compute the cost of relaying and executing the message in advance. The users sending a message through the Synapse Protocol must deposit the tips to get their message to the destination chain.

Anyone can call the `updateGasData` function, which fetches the latest `gasData` from the `Destination` contract and updates the `gasData` for the remote chain with the provided `domain`.

```
function updateGasData(uint32 domain) external {
    (bool wasUpdated, GasData updatedGasData) = _fetchGasData(domain);
    if (wasUpdated) {
        _setGasData(domain, updatedGasData);
    }
}
```

Figure 15.1: The `updateGasData` function in `GasOracle.sol`

Notaries send attestations containing `gasData` to the `Destination` contract. This `gasData` is saved in the `Destination` contract, which is then fetched by the `GasOracle` contract.

The `GasOracle` contract enforces an optimistic delay of five minutes when increasing the gas parameters and one hour when decreasing the parameters:

```
function _updateGasParameter(Number current, Number incoming, uint256 dataMaturity)
    internal
    pure
    returns (Number updatedParameter)
{
    // We apply the incoming value only if its optimistic period has passed.
    // The optimistic period is smaller when the the value is increasing, and bigger
when it is decreasing.
    if (incoming.decompress() > current.decompress()) {
        return dataMaturity < GAS_DATA_INCREASED_OPTIMISTIC_PERIOD ? current :
incoming;
    } else {
        return dataMaturity < GAS_DATA_DECREASED_OPTIMISTIC_PERIOD ? current :
incoming;
    }
}
```

Figure 15.2: The `_updateGasParameter` function in `GasOracle.sol`

The notaries can submit an attestation with a prohibitively high value of gas parameters to store the malicious `gasData` in the `Destination` contract. The `gasData` with high-value parameters will then be fetched by the `GasOracle` contract from the `updateGasData` function, and the `gasData` for the remote chain will be updated five minutes after the submission of the attestation containing malicious `gasData`. This can potentially stop users from sending messages to a destination chain for at least an hour because the gas parameters can be decreased only after an hour after accepting a new attestation.

Exploit Scenario

Eve, a notary, submits an attestation with a malicious snapshot root with the gas parameters value of 2^248. Five minutes after submitting the malicious attestation, assuming no other attestation from the Polygon chain has been submitted, Eve calls the `updateGasData` function of the `GasOracle` contract to set the gas parameters for the Polygon chain to the high values. Because of the high amount of tips required to send the message, users cannot send messages to the Polygon chain.

Recommendations

Short term, enforce a higher delay when incrementing the gas parameters on the destination chain to allow guards to detect malicious attestations.

Long term, review all the delays enforced for various system state updates and ensure that malicious actors cannot update states without being detected.

16. Lack of incentives to update gasData to a lower value

Severity: **Low**	Difficulty: **High**
Type: Data Validation	Finding ID: TOB-PSC-16
Target: `GasOracle.sol`	

Description

The system uses the gas prices of the destination chain to calculate the required tips to send a base message to that blockchain. Because the notaries, guards, and executors earn profit from the `delta` of the actual gas price and the saved gas price, they are disincentivized from ever updating the gas price to a lower value.

Whenever a notary submits an attestation to the `LightInbox` contract, they must also provide the gas information for that particular blockchain. This is saved in the `Destination` contract state and can later be used to update the gas price and calculate the tip amounts. The gas data for a particular destination blockchain can be updated by using the `updateGasData` function of the `GasOracle` contract:

```
function updateGasData(uint32 domain) external {
    (bool wasUpdated, GasData updatedGasData) = _fetchGasData(domain);
    if (wasUpdated) {
        _setGasData(domain, updatedGasData);
    }
}
```

Figure 16.1: The `updateGasData` function in `GasOracle.sol`

This function calls the internal `_fetchGasData` function to fetch the latest data from the `Destination` contract and then update it using the `_setGasData` function. The `_fetchGasData` function checks that the data was not submitted in the same block as the call to `updateGasData`, as shown in figure 16.2.

```
function _fetchGasData(uint32 domain) internal view returns (bool wasUpdated,
GasData updatedGasData) {
    GasData current = _gasData[domain];
    // Destination only has the gas data for the remote domains.
    if (domain == localDomain) return (false, current);
    (GasData incoming, uint256 dataMaturity) =
InterfaceDestination(destination).getGasData(domain);
    // Zero maturity means that either there is no data for the domain, or it was
just updated.
    // In both cases, we don't want to update the local data.
    if (dataMaturity == 0) return (false, current);
    // Update each gas parameter separately.
    updatedGasData = GasDataLib.encodeGasData({
        gasPrice_: _updateGasParameter(current.gasPrice(), incoming.gasPrice(),
dataMaturity),
        dataPrice_: _updateGasParameter(current.dataPrice(), incoming.dataPrice(),
dataMaturity),
        execBuffer_: _updateGasParameter(current.execBuffer(),
incoming.execBuffer(), dataMaturity),
        amortAttCost_: _updateGasParameter(current.amortAttCost(),
incoming.amortAttCost(), dataMaturity),
        etherPrice_: _updateGasParameter(current.etherPrice(),
incoming.etherPrice(), dataMaturity),
        markup_: _updateGasParameter(current.markup(), incoming.markup(),
dataMaturity)
    });
    wasUpdated = GasData.unwrap(updatedGasData) != GasData.unwrap(current);
}
```

Figure 16.2: The `_fetchGasData` function in `GasOracle.sol`

Because the `dataMaturity` variable depends on the `block.timestamp` variable when the attestation was submitted, notaries have some control over this parameter. Since the notaries, guards, and executors earn more tips with a bigger `delta` between the actual gas price and the saved gas price, they could collude to keep the gas price as high as possible by front-running the call to `updateGasData` with a new attestation, causing users to overpay when sending messages.

Similarly, there is no incentive for others to call the `updateGasData` function to update the gas parameters to a lower value, even if notaries submit the lower gas value with the attestation.

Exploit Scenario

The system is deployed on the Synapse chain and Ethereum, and Alice and Bob are assigned as agents. Alice is a notary and Bob is a guard. Alice and Bob decide to collude to keep the gas price data as high as possible by front-running any call to `updateGasData`.

Recommendations

Short term, consider redesigning the way gas data is propagated to minimize the influence that certain system actors have over deciding when gas data updates happen. Otherwise, consider adding an additional incentive for keeping the gas data up to date.

Long term, carefully review the incentive alignment for all system actors to ensure the proper functioning of the Synapse Protocol. Consider creating a list of all parameters that are in control of, or can be influenced by, a particular agent, and ensure their incentives align with the desired system behavior.

17. Risks of executing a base message on a non-recipient contract

Severity: **Low**	Difficulty: **Low**
Type: Data Validation	Finding ID: TOB-PSC-17
Target: `ExecutionHub.sol`	

Description

If the recipient of a base message does not adhere to the `IMessageRecipient` interface but does implement a fallback function, the execution of the message could succeed and potentially lead to unexpected results.

The `Origin` contract allows anyone to send a base message to an arbitrary recipient on another blockchain by calling the `sendBaseMessage` function:

```
function sendBaseMessage(
    uint32 destination,
    bytes32 recipient,
    uint32 optimisticPeriod,
    uint256 paddedRequest,
    bytes memory content
) external payable returns (uint32 messageNonce, bytes32 messageHash) {
    // Check that content is not too large
    if (content.length > MAX_CONTENT_BYTES) revert ContentLengthTooBig();
    // This will revert if msg.value is lower than value of minimum tips
    Tips tips = _getMinimumTips(destination, paddedRequest,
content.length).matchValue(msg.value);
    Request request = RequestLib.wrapPadded(paddedRequest);
    // Format the BaseMessage body
    bytes memory body = BaseMessageLib.formatBaseMessage({
        sender_: msg.sender.addressToBytes32(),
        recipient_: recipient,
        tips_: tips,
        request_: request,
        content_: content
    });
    // Send the message
    return _sendMessage(destination, optimisticPeriod, MessageFlag.Base, body);
}
```

Figure 17.1: The sendBaseMessage function in `Origin.sol`

Once the required verification work is done by the guards and notaries and the attestation is propagated to the `Destination` contract on the receiving blockchain, the message can be executed. This is done by calling the `execute` function of the `Destination` contract, which it inherits from the ExecutionHub contract. If the message is a base message, it will be executed through the internal _executeBaseMessage function.

```
function _executeBaseMessage(Header header, uint256 proofMaturity, uint64 gasLimit,
BaseMessage baseMessage)
    internal
    returns (bool)
{
    // Check that gas limit covers the one requested by the sender.
    // We let the executor specify gas limit higher than requested to guarantee the
execution of
    // messages with gas limit set too low.
    Request request = baseMessage.request();
    if (gasLimit < request.gasLimit()) revert GasLimitTooLow();
    // TODO: check that the discarded bits are empty
    address recipient = baseMessage.recipient().bytes32ToAddress();
    // Forward message content to the recipient, and limit the amount of forwarded
gas
    if (gasleft() <= gasLimit) revert GasSuppliedTooLow();
    try IMessageRecipient(recipient).receiveBaseMessage{gas: gasLimit}({
        origin: header.origin(),
        nonce: header.nonce(),
        sender: baseMessage.sender(),
        proofMaturity: proofMaturity,
        version: request.version(),
        content: baseMessage.content().clone()
    }) {
        return true;
    } catch {
        return false;
    }
}
```

Figure 17.2: The _executeBaseMessage function in ExecutionHub.sol

The _executeBaseMessage function uses a try-catch block to attempt to execute the receiveBaseMessage function on the receiver contract. However, if the contract does not implement the receiveBaseMessage function but does implement a fallback function, the call could succeed and lead to potentially unexpected results.

Exploit Scenario

A protocol team integrates their contracts with the Synapse Protocol and incorrectly assumes that only contracts that implement the IMessageRecipient interface can be called by the Destination contract to execute base messages. If the decentralized application (dapp) grants some special privileges to the Recipient contract, then a fallback function can be used to execute an attack using the Recipient contract's privileges.

Recommendations

Short term, document this behavior in the developer-facing documentation to ensure that developers are aware that messages can be successfully executed on any contract that implements a fallback function.

Long term, explicitly specify preconditions and postconditions for all functions to more easily identify what is being checked and what needs to be checked in a function. Set up fuzzing tests with Echidna to identify unexpected behavior.

18. Front-running risks in functions rewarding the caller	
Severity: **Medium**	Difficulty: **Low**
Type: Timing	Finding ID: TOB-PSC-18
Target: `StatementInbox.sol`, `Inbox.sol`, `BondingManager.sol`, `LightManager.sol`	

Description

The Synapse Protocol rewards the caller of several functions for ensuring timely updates of the system state. An attacker can front-run all these functions to capture the rewards without doing any work. This can discourage guards and notaries from doing the required work for the proper functioning of the Synapse Protocol.

The protocol requires guards and notaries to monitor smart contracts on one chain and update smart contract states on other chains to relay user messages. Most of these functions require a signature from the guard or notary, but some of them reward the caller of the function instead of the signer:

1. The `verifyStateWithSnapshot` function in the `StatementInbox` contract
2. The `verifyStateWithSnapshotProof` function in the `StatementInbox` contract
3. The `verifyStateWithAttestation` function in the `StatementInbox` contract
4. The `verifyStateReport` function in the `StatementInbox` contract
5. The `verifyReceipt` function in the `StatementInbox` contract
6. The `verifyReceiptReport` function in the `StatementInbox` contract
7. The `verifyAttestation` function in the `Inbox` contract
8. The `verifyAttestationReport` function in the `Inbox` contract
9. The `completeSlashing` function in the `BondingManager` contract
10. The `updateAgentStatus` function in the `LightManager` contract

Any user can front-run these functions when called by a guard or notary to capture the reward that was supposed to be paid to the guard or notary for monitoring the protocol smart contracts. This will result in guards and notaries being unable to profit from relaying the messages, and they will eventually stop working for the protocol, potentially halting the protocol or allowing malicious activity from users.

Exploit Scenario

Eve sets up a bot to front-run all the functions listed above and successfully captures all the rewards from the execution of these functions. The guards and notaries who initiated these functions from their off-chain components are unable to recover expenses from running these off-chain components and will eventually stop working for the protocol.

Recommendations

Short term, document the front-running risks of the listed functions to educate the guards and notaries. Encourage them to use front-running–protected remote procedure call (RPC) endpoints for sending these transactions.

Long term, carefully review the incentive alignment for all the system actors to ensure the proper functioning of the Synapse Protocol. Ensure that actors doing the work are rewarded for their work.

19. Use of outdated third-party dependencies	
Severity: **Informational**	Difficulty: **Low**
Type: Patching	Finding ID: TOB-PSC-19
Target: All the smart contracts	

Description

We used `yarn audit` to detect the use of outdated dependencies in the Synapse Protocol codebase and identified multiple vulnerable packages referenced by `yarn.lock`.

Table 19.1 lists the summary of outdated dependencies used in the Synapse Protocol smart contracts (full list in appendix H):

Outdated smart contract dependencies			
Dependencies	**Version used**	**Latest release**	**Vulnerabilities**
`@openzeppelin/contracts-upgradeable`	4.6.0	4.9.1	Five high-severity and four moderate-severity issues
`@openzeppelin/contracts`	4.7.3	4.9.1	One high-severity and two moderate-severity issues

Table 19.1: Outdated dependencies

Table 19.2 lists the vulnerabilities that might affect the Synapse Protocol smart contracts:

All contracts			
Dependencies	**Vulnerability Report**	**Vulnerability Description**	**Vulnerable Versions**
`@openzeppelin/contracts-upgradeable`	TransparentUpgradeableProxy selector clashing	Affected versions of this package have a heightened risk of permanent denial of service for a specific feature.	>= 3.2.0 < 4.8.3

Table 19.2: Vulnerable dependencies that could affect the protocol

In many cases, the use of a vulnerable dependency does not necessarily mean the application is vulnerable. Vulnerable methods from such packages must be called within a particular (exploitable) context. To determine whether Synapse Protocol is vulnerable to these issues, each issue will have to be manually triaged.

Recommendations

Short term, update build process dependencies to their latest versions wherever possible. Use tools such as `retire.js`, `node audit`, and `yarn audit` to confirm that no vulnerable dependencies remain.

Long term, implement these checks as part of the CI/CD pipeline of application development. Do not allow builds to continue with any outdated dependencies.

20. Insufficient event generation	
Severity: **Informational**	Difficulty: **Low**
Type: Auditing and Logging	Finding ID: TOB-PSC-20
Target: All the smart contracts	

Description

Multiple state-changing operations do not emit events. As a result, it will be difficult to monitor the correct behavior of the contracts once they have been deployed, which can result in an unexpected update going unnoticed

Events generated during contract execution aid in monitoring, baselining behavior, and detecting suspicious activity. Without events, users and blockchain-monitoring systems cannot easily detect behavior that falls outside the baseline conditions; malfunctioning contracts and attacks could go undetected.

The following operations should trigger events:

- `setSummitTip`
- `withdrawTips`
- `remoteWithdrawTips`

Exploit Scenario

An attacker discovers a vulnerability in the system that allows them to withdraw all the tips. Because these actions generate no events, the behavior goes unnoticed until there is follow-on damage, such as financial loss.

Recommendations

Short term, add events for all operations that could contribute to a higher level of monitoring and alerting.

Long term, consider using a blockchain-monitoring system to track any suspicious behavior in the contracts. The system relies on several contracts to behave as expected. A monitoring mechanism for critical events would quickly detect any compromised system components.

21. Unsafe cast can be problematic

Severity: **Informational**	Difficulty: **Undetermined**
Type: Data Validation	Finding ID: TOB-PSC-21
Target: All the smart contracts	

Description

The codebase contains unsafe casts that could cause mathematical errors if they are reachable in certain states.

Examples of possible unsafe casts are shown in the figures below:

```
function _nextNonce() internal view returns (uint32) {
    return uint32(_originStates.length);
}
```

Figure 21.1: The `_nextNonce` function in `StateHub.sol`

```
function _saveAttestation(Attestation att, uint32 notaryIndex, uint256 sigIndex)
internal {
    bytes32 root = att.snapRoot();
    if (_rootData[root].submittedAt != 0) revert DuplicatedSnapshotRoot();
    _rootData[root] = SnapRootData({
        notaryIndex: notaryIndex,
        attNonce: att.nonce(),
        attBN: att.blockNumber(),
        attTS: att.timestamp(),
        index: uint32(_roots.length),
        submittedAt: uint40(block.timestamp),
        sigIndex: sigIndex
    });
    _roots.push(root);
}
```

Figure 21.2: The `_saveAttestation` function in `ExecutionHub.sol`

Recommendations

Short term, consider one of the following:

1. Review the codebase to identify all casts that may be unsafe. Analyze whether these casts could be a problem in the current codebase and, if they are unsafe, make the necessary changes to make them safe.

2. Use a `SafeCast` contract from the OpenZeppelin library for all casting operations.

Long term, when implementing potentially unsafe casts, always include comments to explain why those casts are safe in the context of the codebase.

22. Using msg.value in a loop is unsafe

Severity: **Low**	Difficulty: **High**
Type: Data Validation	Finding ID: TOB-PSC-22
Target: `PingPongClient.sol, MessageRecipient.sol`	

Description

When `msg.value` is used in an internal or private function call, and this function is used in a loop, `msg.value` will have the same value through all iterations of the loop. Such a function can be used to drain the contract maliciously.

The `PingPongClient` contract implements the `doPings` function, which executes the `_ping` internal function in a loop:

```
function doPings(uint16 pingCount, uint32 destination_, address recipient, uint16
counter) external {
    for (uint256 i = 0; i < pingCount; ++i) {
        _ping(destination_, recipient.addressToBytes32(), counter);
    }
}
```

Figure 22.1: The doPings function in `PingPongClient.sol`

The `_ping` function makes an internal call to the `_sendMessage` function, which uses the `_sendBaseMessage` function to send the base message:

```
function _ping(uint32 destination_, bytes32 recipient, uint16 counter) internal {
    uint256 pingId = pingsSent++;
    _sendMessage(destination_, recipient, PingPongMessage({pingId: pingId, isPing:
true, counter: counter}));
    emit PingSent(pingId);
}
```

Figure 22.2: The `_ping` *function in* `PingPongClient.sol`

```
function _sendBaseMessage(
    uint32 destination_,
    bytes32 recipient,
    uint32 optimisticPeriod,
    MessageRequest memory request,
    bytes memory content
) internal returns (uint32 messageNonce, bytes32 messageHash) {
    if (recipient == 0) revert IncorrectRecipient();
    return InterfaceOrigin(origin).sendBaseMessage{value: msg.value}(
        destination_, recipient, optimisticPeriod, _encodeRequest(request), content
    );
}
```

Figure 22.3: The `_sendBaseMessage` function in `MessageRecipient.sol`

Since the `_sendBaseMessage` function is executed in a loop, the value of `msg.value` will stay the same through all iterations. Although the impact of this issue is limited in the current implementation because the `doPings` function is not payable, a protocol team could use this as a reference implementation and potentially introduce vulnerabilities in their code.

Exploit Scenario

A protocol team uses the `PingPongClient` and `MessageRecipient` contracts as a reference implementation for their own client and implements the same looping mechanism. Eve notices this and uses the loop to inflate her balance and steal assets from the protocol.

Recommendations

Short term, consider the following changes to mitigate the risks:

1. Mark the `doPings` and `doPing` functions as payable.
2. Update the `_sendBaseMessage` internal function to take the value of native tokens to be transferred as an argument, instead of using `msg.value`. This will make the use of the value explicit and minimize the risks of unsafe use of this function.

Long term, use the Slither static analyzer to catch common issues such as this one. Consider integrating a Slither scan into the project's CI pipeline, pre-commit hooks, or build scripts.

A. Vulnerability Categories

The following tables describe the vulnerability categories, severity levels, and difficulty levels used in this document.

Vulnerability Categories	
Category	Description
Access Controls	Insufficient authorization or assessment of rights
Auditing and Logging	Insufficient auditing of actions or logging of problems
Authentication	Improper identification of users
Configuration	Misconfigured servers, devices, or software components
Cryptography	A breach of system confidentiality or integrity
Data Exposure	Exposure of sensitive information
Data Validation	Improper reliance on the structure or values of data
Denial of Service	A system failure with an availability impact
Error Reporting	Insecure or insufficient reporting of error conditions
Patching	Use of an outdated software package or library
Session Management	Improper identification of authenticated users
Testing	Insufficient test methodology or test coverage
Timing	Race conditions or other order-of-operations flaws
Undefined Behavior	Undefined behavior triggered within the system

Severity Levels	
Severity	**Description**
Informational	The issue does not pose an immediate risk but is relevant to security best practices.
Undetermined	The extent of the risk was not determined during this engagement.
Low	The risk is small or is not one the client has indicated is important.
Medium	User information is at risk; exploitation could pose reputational, legal, or moderate financial risks.
High	The flaw could affect numerous users and have serious reputational, legal, or financial implications.

Difficulty Levels	
Difficulty	**Description**
Undetermined	The difficulty of exploitation was not determined during this engagement.
Low	The flaw is well known; public tools for its exploitation exist or can be scripted.
Medium	An attacker must write an exploit or will need in-depth knowledge of the system.
High	An attacker must have privileged access to the system, may need to know complex technical details, or must discover other weaknesses to exploit this issue.

B. Code Maturity Categories

The following tables describe the code maturity categories and rating criteria used in this document.

Code Maturity Categories	
Category	Description
Arithmetic	The proper use of mathematical operations and semantics
Auditing	The use of event auditing and logging to support monitoring
Authentication / Access Controls	The use of robust access controls to handle identification and authorization and to ensure safe interactions with the system
Complexity Management	The presence of clear structures designed to manage system complexity, including the separation of system logic into clearly defined functions
Decentralization	The presence of a decentralized governance structure for mitigating insider threats and managing risks posed by contract upgrades
Documentation	The presence of comprehensive and readable codebase documentation
Front-Running Resistance	The system's resistance to front-running attacks
Low-Level Manipulation	The justified use of inline assembly and low-level calls
Testing and Verification	The presence of robust testing procedures (e.g., unit tests, integration tests, and verification methods) and sufficient test coverage

Rating Criteria	
Rating	Description
Strong	No issues were found, and the system exceeds industry standards.
Satisfactory	Minor issues were found, but the system is compliant with best practices.
Moderate	Some issues that may affect system safety were found.
Weak	Many issues that affect system safety were found.
Missing	A required component is missing, significantly affecting system safety.

Not Applicable	The category is not applicable to this review.
Not Considered	The category was not considered in this review.
Further Investigation Required	Further investigation is required to reach a meaningful conclusion.

C. Code Quality Findings

This appendix lists findings that are not associated with specific vulnerabilities.

- **Constant variables with expression assignment**
  - The SNAPSHOT_MAX_STATES and TIPS_MULTIPLIER constants are assigned to expressions that involve other constants.
  - If the expressions result in a revert (e.g., from overflow), it will cause every function that uses the constants to always revert.
  - Instead of assigning expressions to constants, assign values.
- **Incorrect comments**
  - The NatSpec comment on the `BaseMessageLib` library mentions an incorrect `request` length in BaseMessage.sol.
  - The NatSpec comment on the `SnapshotLib` library mentions an incorrect `state` length in Snapshot.sol.
  - The NatSpec comments on the `acceptGuardSnapshot`, `acceptNotarySnapshot`, and `acceptReceipt` functions in the `Summit` contract are incorrect. The comments say the functions are only callable by the `AgentManager` contract instead of the `Inbox` contract in InterfaceSummit.sol.
- **Unused code**
  - The `Origin` and `StateHub` contracts extend the `AgentSecured` contract for the `onlyAgentManager` modifier. All the other code in the `AgentSecure` contract that manages dispute status is not used in the `Origin` contract. The `Origin` contract can implement the `onlyAgentManager` modifier itself instead of extending the `AgentSecured` contract.
- **Unused variables**
  - The `msgOrigin` variable in the `remoteSlashAgent` function in BondingManager.sol is unused.
  - The `sigIndex` variable in the `_saveReceipt` function in Summit.sol is unused.
- There is no zero-value check in the `addAgent` function in BondingManager.sol.

D. Automated Analysis Tool Configuration

Slither

We used Slither to detect common issues and anti-patterns in the codebase. Slither discovered multiple low- and medium-severity issues, such as TOB-PSC-02 and TOB-PSC-22. Integrating Slither into the project's testing environment can help find similar issues and improve the overall quality of the smart contracts' code.

```
slither --filter-paths “test” --foundry-out-directory artifacts .
```

Figure D.1: The command used to run `slither-analyzer`

Integrating `slither-action` into the project's CI pipeline can automate this process.

Echidna

We used Echidna, a smart contract fuzzer that can rapidly test security properties via malicious, coverage-guided test case generation, to check various system states. Echidna uncovered issues such as TOB-PSC-14. These issues would have been difficult to detect if relying solely on manual analysis.

```
function test_isValidReceipt(uint256 messageIndex) public view {
    require(messages.length != 0);
    messageIndex = messageIndex % messages.length;
    Messages memory message = messages[messageIndex];
    bytes memory rcptPayload = receiptPayloads[message.msgHash];
    require(message.executed && rcptPayload.length != 0);

    assert(destination.isValidReceipt(rcptPayload));
}
```

Figure D.2: An invariant implementation in the Echidna test harness

```
deployer: "0x30000"
sender: ["0x10000", "0x20000", "0x30000"]
corpusDir: "contracts/echidna/corpus"
testMode: "assertion"
testLimit: 500000
codeSize: 0x6000000
```

Figure D.3: Sample configuration setup in `config.yaml`

```
echidna . --contract EchidnaSynapse --config config.yaml
```

Figure D.4: The command used to run the Echidna fuzzing campaign

For further guidance on using Echidna, including in-depth documentation and practical examples, visit our Building Secure Contracts website or watch our Echidna live streams, available on YouTube.

E. Try-Catch Statement Considerations

The try-catch syntax introduces new capabilities in Solidity that allow a contract to more easily react to a failure or error in the callee (the smart contract being called) without causing a revert in the currently executing contract. Before the try-catch feature was added, this was possible only by using low-level calls. This feature can be used with external function calls and contract creation calls. This appendix outlines the most important caveats to keep in mind while developing a smart contract system that uses this feature.

General Considerations

❏ **Public and external functions of the currently executing contract are valid targets of the `try` clause**. There are multiple ways this can be achieved (e.g., by using `this.functionName` or wrapping the contract address in an interface), but they will all behave the same as any external call—i.e., the `msg.sender` will be the caller (in this case, the contract itself), and the attached value could come from the caller's balance.

❏ **The call can trigger the target contract's fallback function**. If the target contract does not implement the function being called but does implement a fallback function, the call could succeed and trigger the `try` clause body.

❏ **Calls to externally owned accounts (EOAs) will not trigger the `catch` clause but will instead cause a revert in the currently executing contract.** This is because the compiler adds an implicit contract existence check before the try-catch block is reached. One way to mitigate this issue is to add an explicit contract existence check in the code so that if the receiver is not a contract, a different code block can be executed without reverting.

❏ **Local variables defined in either the `try` or `catch` clause are inaccessible outside these clauses.**

❏ **Any reverts inside either of the clauses or the function parameters will not be caught but will instead revert the current context.**

❏ **The `catch` clause body can be forcibly triggered by providing a lower gas amount, such that the external call runs out of gas**. Whenever an external call is executed, 63/64th of the gas amount is provided for the execution of that call, and 1/64th is kept for the continued execution of the current context. A malicious user can force the call to fail (and trigger the `catch` clause) by providing a gas amount that will cause the external call to run out of gas but still be enough to finish execution in the parent context. This can also happen if the amount of gas provided to the external call is a user-defined variable. To mitigate this issue, the exact revert reason can be checked in the `catch` clause body.

- ❏ **As with any external call, try-catch can lead to reentrancy issues**. Try-catch can be thought of as syntactic sugar for a low-level call, so it has many of the same risks as a low-level call. To mitigate this issue, the contract should follow the Checks-Effects-Interactions pattern or use a mutex like OpenZeppelin's `ReentrancyGuard`.

Return Values and Error Handling

- ❏ **If the `try` clause expects return data but the function being called does not return any data, the current context will revert without triggering the `try` or `catch` clause body**. If the system expects the current context to never revert, this can potentially lead to a denial-of-service attack.

- ❏ **If the returned data does not match the expected return type, the `try` clause body could still be triggered.** As long as the returned data can be decoded into the expected type, the `try` clause body will be triggered. For example, if the `try` clause expects a `uint8` and the function returns a `uint256` type that can fit into a `uint8`, the current context will not revert. More surprisingly, if the `try` clause expects a `bytes32` and the function returns a `bytes` type, the value will be successfully decoded as the offset of the `bytes` data.

- ❏ **If the returned data does not match the expected return type and it cannot be decoded to fit into this return type, the current context will be reverted**. For example, if the `try` block expects a `uint8` type and the function returns a `uint256` type larger than 255, the current context will revert.

- ❏ **If the target contract is compiled with an older compiler version (earlier than 0.8.0), Panic errors cannot be caught with `catch Panic(uint errorCode) {}`.** This is because Panic errors were introduced in 0.8.0; it is an invalid opcode in lesser versions.

- ❏ **Errors will bubble up from most external calls, so an error caught with the `catch` clause will not necessarily originate from the contract being called**. It could originate from any other call the target contract is making.

F. Rounding Recommendations

The money market fund's arithmetic rounds down on every operation, which can lead to a loss of precision that benefits the user instead of the system. This was the root cause of reported issues (TOB-PSC-07 and TOB-PSC-10).

The following guidelines describe how to determine the rounding direction per operation. We recommend applying the same analysis for all arithmetic operations.

Determining Rounding Direction

To determine how to apply rounding (whether up or down), consider the result of the expected output.

For example, consider the formula for calculating the minimum tip amount:

```
	summitTip = (_summitTipWei << NumberLib.BWAD_SHIFT) /
localEtherPrice;
```

To benefit the protocol, the rounding direction must tend towards higher values to maximize the amount sent by a user with a message. Therefore, it should round up.

Similar rounding techniques can be applied to all the system's formulas to ensure that rounding always occurs in the direction that benefits the protocol.

Rounding Rules

- When funds leave the system, these values should round down to favor the protocol over the user. Rounding these values up can allow attackers to profit from the rounding direction by receiving more than intended on fund interactions.
- When funds enter the system, these values should always round up to maximize the number of tokens the protocol receives. Rounding these values down can result in near-zero values, which can allow attackers to profit from the rounding direction by receiving heavily discounted funds. Rounding down to zero can allow attackers to steal funds.
- If the result of the computation can be positive or negative, then the **r**ounding direction must mirror the sign of the result to benefit the protocol.

Recommendations for Further Investigation

- Create unit tests to highlight the result of the precision loss. Unit tests will help validate the manual analysis.
- Use fuzzing and differential testing to find rounding issues.

G. Incident Response Recommendations

This appendix provides recommendations on formulating an incident response plan.

- **Identify the parties (either specific people or roles) responsible for implementing the mitigations when an issue occurs (e.g., deploying smart contracts, pausing contracts, upgrading the front end, etc.).**
- **Document internal processes for addressing situations in which a deployed remedy does not work or introduces a new bug.**
    - Consider documenting a plan of action for handling failed remediations.
- **Clearly describe the intended contract deployment process.**
- **Outline the circumstances under which** the team **will compensate users affected by an issue (if any).**
    - Issues that warrant compensation could include an individual or aggregate loss or a loss resulting from user error, a contract flaw, or a third-party contract flaw.
- **Document how the team plans to stay up to date on new issues that could affect the system; awareness of such issues will inform future development work and help the team secure the deployment toolchain and the external on-chain and off-chain services that the system relies on.**
    - Identify sources of vulnerability news for each language and component used in the system, and subscribe to updates from each source. Consider creating a private Discord channel in which a bot will post the latest vulnerability news; this will provide the team with a way to track all updates in one place. Lastly, consider assigning certain team members to track news about vulnerabilities in specific system components.
- **Determine when the team will seek assistance from external parties (e.g., auditors, affected users, other protocol developers) and how it will onboard them.**
    - Effective remediation of certain issues may require collaboration with external parties.
- **Define contract behavior that would be considered abnormal by off-chain monitoring solutions.**

It is best practice to perform periodic dry runs of scenarios outlined in the incident response plan to find omissions and opportunities for improvement and to develop "muscle memory." Additionally, document the frequency with which the team should perform dry runs of various scenarios, and perform dry runs of more likely scenarios more regularly. Create a template to be filled out with descriptions of any necessary improvements after each dry run.

H. Outdated Dependencies

In this appendix, we list all the outdated dependencies and the associated vulnerabilities.

All contracts			
Dependencies	**Vulnerability Report**	**Vulnerability Description**	**Vulnerable Versions**
`@openzeppelin/contracts-upgradeable` `@openzeppelin/contracts`	`TransparentUpgradeableProxy` selector clashing	Affected versions of this package have a heightened risk of permanent denial of service (DoS) for a specific feature.	>= 3.2.0 < 4.8.3
`@openzeppelin/contracts-upgradeable`	`SignatureChecker` may revert on invalid EIP-1271 signers	Affected versions of this package could cause incorrect handling of invalid EIP-1271 signatures.	>= 4.1.0 < 4.7.1
`@openzeppelin/contracts-upgradeable`	`GovernorVotesQuorumFraction` updates to quorum may affect past defeated proposals	Affected versions of this package could cause failed governance proposals to become executable.	>= 4.3.0 < 4.7.2
`@openzeppelin/contracts-upgradeable`	`ERC165Checker` may revert instead of returning `false`	Affected versions of this package could cause functions that use `ERC165Checker` to unexpectedly revert.	>= 4.0.0 < 4.7.1
`@openzeppelin/contracts-upgradeable`	ECDSA signature malleability	Affected versions of this package could lead to signature replay attacks.	>= 4.1.0 < 4.7.3

`@openzeppelin/contracts-upgradeable`	Cross-chain utilities for Arbitrum L2 see EOA calls as cross-chain calls	Affected versions of this package could misclassify direct interactions from EOAs as cross-chain calls.	>= 4.6.0 < 4.7.2
`@openzeppelin/contracts-upgradeable`	`ERC165Checker` unbounded gas consumption	Affected versions of this package could pose a DoS risk.	>= 3.2.0 < 4.7.2

`@openzeppelin/contracts-upgradeable`
 	
`@openzeppelin/contracts`
	`GovernorCompatibilityBravo` may trim proposal calldata	Affected versions of this package could cause incorrect reporting of proposal calldata and lead to a proposal executing in unexpected ways.	>= 3.2.0
			
			< 4.8.3

`@openzeppelin/contracts-upgradeable`
 	
`@openzeppelin/contracts`
	Governor proposal creation may be blocked by front-running	Affected versions of this package could introduce front-running risks during the creation of a proposal.	>= 4.3.0
			
			< 4.9.1