Wonderland Prophet

Security Assessment

October 18, 2024

Prepared for:
Gas
Wonderland

Prepared by: **Elvis Skoždopolj, Tjaden Hess, and Nat Chin**

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.
497 Carroll St., Space 71, Seventh Floor
Brooklyn, NY 11215
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

Project Summary

Contact Information

The following project manager was associated with this project:

Sam Greenup, Project Manager
sam.greenup@trailofbits.com

The following engineering director was associated with this project:

Josselin Feist, Engineering Director, Blockchain
josselin.feist@trailofbits.com

The following consultants were associated with this project:

Elvis Skoždopolj, Consultant
elvis.skozdopolj@trailofbits.com

Nat Chin, Consultant
nat.chin@trailofbits.com

Tjaden Hess, Consultant
tjaden.hess@trailofbits.com

Project Timeline

The significant events and milestones of the project are listed below.

Date	Event
April 26, 2024	Pre-project kickoff call
May 10, 2024	Status update meeting #1
May 17, 2024	Delivery of report draft
May 28, 2024	Report readout meeting
October 18, 2024	Delivery of comprehensive report with fix review

Executive Summary

Engagement Overview

Wonderland engaged Trail of Bits to review the security of its modular oracle smart contracts, including `prophet-core` and `prophet-modules`. The core contracts define the core oracle logic and the base module implementation, while the module contracts consist of plug-and-play dispute, response, finality, resolution, and request modules.

A team of three consultants conducted the review from May 6 to May 17, 2024, for a total of four engineer-weeks of effort. Our testing efforts focused on identifying issues related to front-running exploits, assessing whether expected control flows can be bypassed and whether proper access controls are enforced, and identifying concerns related to manipulated oracle results. We focused on exploits that would result in loss of funds, funds getting stuck, or could cause unauthorized or incorrect oracle data responses. We also reviewed implementation complexity, logical edge cases, and the ability for higher level protocols to securely develop modules and integrate with the Prophet protocol.

With full access to source code and documentation, we performed static and dynamic testing of the prophet-core and prophet-modules using automated and manual processes. While we reviewed both the core contracts and the module contracts, the core contracts were considered a higher priority during this review.

Observations and Impact

In developing the target codebases, DeFi Wonderland has prioritized modularity and support for multiple use cases; this focus has increased the codebases' complexity and related risks across most of the components. Based on the quantity of vulnerabilities present in the first-party example modules, it is clear that there are not sufficient guardrails and guidance in place for third-party developers to securely create new modules for integration into the ecosystem.

During our review, we identified many issues related to the following, indicating that the codebases would benefit from improved testing and data structure design:

- Insufficient data validation and consistency checking
- Lack of clarity and expectations in regard to access controls

During the audit, we discovered several significant findings across most of the components, including the following:

- Insufficient data validation in the resolution and dispute modules allowing users to bypass deadlines and manipulate dispute resolution (TOB-WOND-4)

- Ineffective access controls in the accounting extensions allowing a malicious user to drain the entire balance of these extensions (TOB-WOND-5)
- Insufficient data validation in the `RootVerificationModule` leading to users being able to append arbitrary data to their response and the response being considered valid (TOB-WOND-6)
- Insufficient validation of request/response existence, breaking user flow assumptions (TOB-WOND-13)
- Insufficient validation of the dispute struct leading to the wrong user's balance being reduced, and potentially preventing a request from being finalized (TOB-WOND-12)
- A combination of lack of access controls and a hard-coded value in the `claimEscalationReward` function of the `BondEscalationAccounting` contract, enabling any user to steal another user's tokens for a dispute in the `NoResolution` status (TOB-WOND-9)

Our review found that the testing suite is ineffective in discovering these issues due to the heavy use of mock contracts and calls in the unit tests. Furthermore, the module contracts appear to require further development and testing in order to address the access control and data validation issues.

Recommendations

Based on the codebase maturity evaluation and findings identified during the security review, Trail of Bits recommends that Wonderland take the following steps prior to considering deployment:

- **Remediate the findings disclosed in this report.** We found multiple issues that would allow an attacker to drain the protocol, break protocol assumptions, or influence the accounting of the system. The issues stem from a lack of a systematic approach to data validation and access controls. These findings should be addressed as part of a direct remediation or as part of any refactor that may occur when addressing other recommendations.
- **Reduce implicit delegated trust.** Ensure that actions taken on behalf of a user are performed either by the user themselves or by an actor explicitly approved by the user to take that specific action. For example, a user should be marked as the proposer of a response only if that user directly proposed the response or explicitly delegated the ability to do so to the caller. Do not treat modules as implicitly authorized to perform actions on behalf of any user.
- **Write a system specification and re-evaluate the architecture of the system.** The current architecture is fully modular and extremely flexible; however, this is a double-edged sword, since it increases the attack surface of the system.

Retroactively writing a system specification and using it to re-evaluate the system architecture would help guide future development, identify bugs and inconsistencies in the system, prevent bugs from being introduced during development, and improve maintainability. We recommend applying stricter and more well-defined access controls and reducing the attack surface by limiting an Oracle contract to only one set of modules.

- **Improve the testing suite by mirroring the production deployment of the system.** Although using mock contracts and calls can simplify the setup of a testing suite, it increases the risk that development errors will go undetected. One example of this is TOB-WOND-8, for which all of the tests pass, although the functions will always revert in production. Setting up the tests in a way that mirrors the production deployment as closely as possible will enhance the effectiveness of the testing suite, improving the chances of finding issues.

- **Perform another security review before deployment.** The findings disclosed in this report indicate that the architecture of the system could be improved in order to reduce the attack surface and simplify the system. This will require large architectural changes and code refactors, which will necessitate another security review before deploying the system.

Finding Severities and Categories

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

EXPOSURE ANALYSIS

Severity	*Count*
High	6
Medium	4
Low	4
Informational	1
Undetermined	1

CATEGORY BREAKDOWN

Category	*Count*
Access Controls	2
Cryptography	1
Data Validation	8
Denial of Service	5

Project Goals

The engagement was scoped to provide a security assessment of the Wonderland prophet-core and prophet-modules. Specifically, we sought to answer the following non-exhaustive list of questions:

- Is it possible for an attacker to steal funds?
- Does the state machine work as expected?
- Are access controls effective in preventing unauthorized users from calling protected functions?
- Does arithmetic underflow or overflow affect the code?
- Is it possible for a participant to bypass a timelock?
- Are the state updates between the modules and oracle correct and consistent?
- Can users improperly influence any part of the state updates through any means, such as reentrancy?
- Does the system correctly handle various types of ERC20 tokens?
- Is the system susceptible to transaction ordering attacks?
- Can external parties unfairly influence a request, response, or dispute?
- Can a valid request, response, or dispute become stuck?
- Can users' funds become permanently locked in the contracts?
- Does the system safely handle low-level calls?

Project Targets

The engagement involved a review and testing of the targets listed below.

prophet-core

Repository	https://github.com/defi-wonderland/prophet-core
Version	4d4a448
Type	Solidity
Platform	Ethereum

prophet-modules

Repository	https://github.com/defi-wonderland/prophet-modules
Version	`8a5f3f3`
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:

- **Core:** The core contracts consist of the `Oracle` contract, which manages the general request/response/dispute flow and interacts with the modules specified in the request, and the `Module` contract, which is a base contract that all modules inherit from. We manually reviewed both components for incorrect state updates; bypass of the expected user flows that could lead to denial of service or cause invalid data to be considered as valid; issues related to front-running any of the user actions; and access control and data validation issues. We prioritized verifying the core assumptions of the `Oracle`, namely: that actions cannot become stuck or be forcefully considered as valid; that the correct actors are charged the bonds; and that submitted requests/responses/disputes cannot be modified by external actors.

- **Modules:** The modules consist of multiple smart contracts in charge of processing requests, responses, disputes, finalizing requests, and resolving disputes. These contracts are provided as part of the request data, meaning that each request can interact with an arbitrary set of modules. We manually reviewed the module contracts for issues related to data validation and access controls; denial-of-service attack vectors; and any accounting issues that could lead to loss of user assets. Additionally, we reviewed how the modules interact with each other and the Oracle contract, looking for any potentially problematic interactions within the system.

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:

- We prioritized the core contracts over the module contracts, so module interactions and callbacks received a less in-depth review. The escalation modules received less coverage due to containing complex logic and interactions.

- We did not review third-party components, such as OpenZeppelin contracts and the `MerkleLib` library.

Codebase Maturity Evaluation: Prophet Modules

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 module contracts use unchecked arithmetic for many balance adjustments. We found one instance (TOB-WOND-5) in which underflow allows a separate access-control bypass, allowing an attacker to more quickly drain user funds.	**Moderate**
Auditing	All state-changing operations emit events, allowing protocol operators to easily monitor the system. Higher-level protocols using the oracle system should implement monitoring and incident-response protocols.	**Satisfactory**
Authentication / Access Controls	We found serious issues related to access controls and input data authentication, leading to loss of user funds (TOB-WOND-4, TOB-WOND-5). The overall system design is prone to access control violations because accounting contracts grant permissions to arbitrary, dynamically specified dispute and resolution modules. Modules must be carefully selected and explicitly approved by all users of the contract before access to modify global balances is granted. The client should carefully specify and document trust relations and access permissions among the various contracts.	**Weak**
Complexity Management	Most of the module implementations are straightforward and cleanly implemented. The `BondEscalation` contract is complex, with deeply nested conditionals in some functions. Control flow is complex; each user transaction may involve several different contracts, input-dependent call flow, and low-level calls with dynamic targets.	**Moderate**
Cryptography and Key	We did not find any cryptographic issues in the modules.	**Satisfactory**

Management		
Decentralization	All contracts are immutable and user-configurable. Validity and availability of the oracle results depends on the higher choice of resolution module.	**Satisfactory**
Documentation	The client provided clear documentation for most modules and flowcharts for contract interactions. Documentation was lacking with regard to access controls and trust relationships.	**Moderate**
Low-Level Manipulation	Several modules use low-level calls, some of which are missing contract existence checks (TOB-WOND-1). Low-level calls supporting user-chosen data may be dangerous, especially when combined with insufficient access control such as that described in TOB-WOND-5.	**Moderate**
Testing and Verification	All modules have unit tests, but due to the heavy use of mocked contracts in unit tests, further integration tests are necessary to prevent issues like TOB-WOND-8. The accounting extensions are critical components and do not have sufficient tests.	**Moderate**
Transaction Ordering	Many aspects of the protocol are dependent on transaction ordering. External accounts may be unable to use the contract due to race conditions (TOB-WOND-14).	**Weak**

Codebase Maturity Evaluation: Prophet Core

Category	Summary	Result
Arithmetic	The core contracts use unchecked arithmetic only for loop index increments, where overflows are not a concern. There is no rounding or complex arithmetic.	Satisfactory
Auditing	All state-changing operations emit events, allowing protocol operators to easily monitor the system. Higher-level protocols using the oracle system should implement monitoring and incident-response protocols.	Satisfactory
Authentication / Access Controls	The core contracts do not have any privileged actors. We found one issue allowing a malicious dispute module to propose a response on behalf of a user who did not approve the action (TOB-WOND-15).	Satisfactory
Complexity Management	The smart contract state consists of many independent mappings that could be simplified into a single mapping from `requestID` to a struct containing all elements of the request state. The protocol relies heavily on callbacks, which complicates control flow and increases both the attack surface for reentrancy and access control issues. The use of a single contract to store the states for requests from many different higher-level protocols and the use of unrestricted dynamic modules increase complexity and may allow undesirable cross-protocol interactions.	Moderate
Cryptography and Key Management	The computation of request, response, and dispute IDs relies on hashing without domain separation (TOB-WOND-16). The core contracts otherwise do not use cryptographic primitives.	Moderate
Decentralization	The contracts are immutable, and no party has the ability to unilaterally disable the protocol functionality.	Satisfactory
Documentation	The client provided clear documentation for most modules and flowcharts for contract interactions. We encourage improving the documentation around	Satisfactory

	securely implementing modules.	
Low-Level Manipulation	The core contracts use some assembly for efficiency, but it is well documented and self-contained.	**Satisfactory**
Testing and Verification	Unit tests achieve good coverage but rely heavily on mocked contracts. More complex and realistic module contracts in either the unit tests or integration tests would help to ensure that the core contracts behave as expected.	**Satisfactory**
Transaction Ordering	Many aspects of the protocol are dependent on transaction ordering. External accounts may be unable to use the contract due to race conditions (TOB-WOND-14).	**Moderate**

Summary of Findings

The tables below summarize the findings of the review, including type and severity details.

Prophet Modules

ID	Title	Type	Severity
1	Lack of contract existence check on low-level calls	Data Validation	High
2	Contracts are incompatible with ERC20 tokens with fee	Data Validation	Low
3	Dispute resolution can be blocked due to high gas costs	Data Validation	Low
4	Arbitrary request data can be provided to module functions in order to bypass validation	Data Validation	High
5	Ineffective access control can lead to user funds being stolen	Access Controls	High
6	RootVerificationModule response can contain arbitrary data	Data Validation	Low
7	Executing the resolveDispute and revealVote functions in the same block can lead to loss of funds	Denial of Service	Medium
8	PrivateERC20ResolutionModule functions will always revert	Denial of Service	Medium
9	Dispute pledges for a NoResolution dispute can be stolen by anyone	Denial of Service	High
10	Disputer’s escalation pledge may become locked	Data Validation	Medium
11	Use of low-level calls can compromise access control	Access Controls	High

Prophet Core

ID	Title	Type	Severity
12	The wrong user can be charged for a resolved dispute	Data Validation	**Medium**
13	Insufficient validation of request/response existence could have unintended side effects	Data Validation	**Undetermined**
14	Externally owned accounts may be unable to submit requests during periods of high usage	Denial of Service	**Low**
15	A user’s balance can be bonded against their will	Denial of Service	**High**
16	Lack of domain separation between request, response, and dispute IDs	Cryptography	**Informational**

Detailed Findings: Prophet Modules

1. Lack of contract existence check on low-level calls	
Severity: **High**	Difficulty: **High**
Type: Data Validation	Finding ID: TOB-WOND-1
Target: `prophet-modules/solidity/contracts/modules/*`	

Description

The module contracts use the low-level `call`, which does not have contract existence checks. If the target contract being called is set to an incorrect address or the address of a contract that was destroyed, the low level call will still return "success." This is also applicable to delegatecalls, although these are not used in the respective contracts.

The Solidity documentation includes the following warning:

> The low-level functions `call`, `delegatecall` and `staticcall` return `true` as their first return value if the account called is non-existent, as part of the design of the EVM. Account existence must be checked prior to calling if needed.

Figure 1.1: A snippet of the Solidity documentation detailing unexpected behavior related to `delegatecall`

This can be seen in the `disputeResponse` function in the modules, where the contracts executes a low level call on the `_params.verifier` address, and lacks a contract existence check:

```
/// @inheritdoc ICircuitResolverModule
function disputeResponse(
  IOracle.Request calldata _request,
  IOracle.Response calldata _response,
  IOracle.Dispute calldata _dispute
) external onlyOracle {
  RequestParameters memory _params = decodeRequestData(_request.disputeModuleData);

  (bool _success, bytes memory _correctResponse) =
_params.verifier.call(_params.callData);

  if (!_success) revert CircuitResolverModule_VerificationFailed();
```

Figure 1.2: The `disputeResponse` function in `CircuitResolverModule.sol`

This can also be seen in the `CallbackModule`, where parameters are decoded, then code is executed on the `_params.target` address:

```
/// @inheritdoc ICallbackModule
function finalizeRequest(
  IOracle.Request calldata _request,
  IOracle.Response calldata _response,
  address _finalizer
) external override(Module, ICallbackModule) onlyOracle {
  RequestParameters memory _params = decodeRequestData(_request.finalityModuleData);
  _params.target.call(_params.data);
  emit Callback(_response.requestId, _params.target, _params.data);
  emit RequestFinalized(_response.requestId, _response, _finalizer);
}
```

Figure 1.3: The `finalizeRequest` function in `CallbackModule.sol`

Similarly, in the `MultipleCallbacksModule`, the function loops over all targets with the params data, but does not ensure that there is code at these addresses:

```
/// @inheritdoc IMultipleCallbacksModule
function finalizeRequest(
  IOracle.Request calldata _request,
  IOracle.Response calldata _response,
  address _finalizer
) external override(IMultipleCallbacksModule, Module) onlyOracle {
  RequestParameters memory _params = decodeRequestData(_request.finalityModuleData);
  uint256 _length = _params.targets.length;

  for (uint256 _i; _i < _length;) {
    _params.targets[_i].call(_params.data[_i]);
    emit Callback(_response.requestId, _params.targets[_i], _params.data[_i]);
    unchecked {
      ++_i;
    }
  }
```

Figure 1.4: The `finalizeRequest` function in `MultipleCallbacksModule.sol`

Exploit Scenario
Alice, a user, accidentally sets a request target as an EOA. Due to the lack of contract existence check, the call appears to succeed, and the `Callback` and `RequestFinalized` events are emitted. However, no code ends up being executed.

Recommendations
Short term, implement a contract existence check before each low-level call. Document the fact that using `selfdestruct` can lead to unexpected behavior, and prevent future upgrades from introducing these functions.

Long term, carefully review the Solidity documentation, especially the “Warnings” section, and understand the pitfalls of using low-level calls.

2. Contracts are incompatible with ERC20 tokens with fee

Severity: **Low**	Difficulty: **Undetermined**
Type: Data Validation	Finding ID: TOB-WOND-2
Target: `prophet-modules/solidity/contracts/extensions/AccountingExtension.sol`	

Description

The Prophet contracts are meant to interface with ERC20 tokens. However, several ERC20 tokens do not fully comply with the ERC20 standard, meaning the contract will not work with such tokens.

For example, some ERC20 tokens take a fee on transfer, which means the actual amount received by a contract will not be equal to the amount of tokens sent. As a result, the increasing of the `balanceOf` data structures in the `AccountingExtension` contract may not scale correctly.

```
function deposit(IERC20 _token, uint256 _amount) external {
  _token.safeTransferFrom(msg.sender, address(this), _amount);
  balanceOf[msg.sender][_token] += _amount;
  emit Deposited(msg.sender, _token, _amount);
}

/// @inheritdoc IAccountingExtension
function withdraw(IERC20 _token, uint256 _amount) external {
  uint256 _balance = balanceOf[msg.sender][_token];

  if (_balance < _amount) revert AccountingExtension_InsufficientFunds();

  unchecked {
    balanceOf[msg.sender][_token] -= _amount;
  }

  _token.safeTransfer(msg.sender, _amount);

  emit Withdrew(msg.sender, _token, _amount);
}
```

Figure 2.1: The `deposit` and `withdraw` functions in `AccountingExtension.sol`

Exploit Scenario

Alice, a user, deposits 20 XYZ tokens into the `AccountingExtension`. The token takes a fee on transfer, diverting a small portion of the token to a registry, such that the registry receives 2 XYZ tokens, and the `AccountingExtension` receives 18 XYZ tokens. The `deposit` function above assumes that the amount of tokens that entered the contract is exactly equal to the amount transferred. However, this assumption does not hold, so the `balanceOf[msg.sender][_token]` ends up increasing too much.

Recommendations

Short term, either document the tokens that are not expected to work with the system to ensure that users know what to look for while interacting with the Prophet contracts, or ensure that the amount of token received is equal to the amount that was transferred.

Long term, follow recommendations outlined in our Token Integration Checklist in appendix C.

3. Dispute resolution can be blocked due to high gas costs

Severity: **Low**	Difficulty: **Medium**
Type: Data Validation	Finding ID: TOB-WOND-3
Target: `prophet-modules/solidity/contracts/modules/resolution/{PrivateERC20ResolutionModule.sol, ERC20ResolutionModule.sol}`	

Description

Once a vote has passed, the use of the OpenZeppelin `EnumerableSet`'s `values` function may cause the dispute resolution process to revert and run out of gas. The `PrivateERC20ResolutionModule` uses the `EnumerableSet` to manage votes by copying over a large array from storage to memory.

```
/**
 * @dev Return the entire set in an array
 *
 * WARNING: This operation will copy the entire storage to memory, which can be
quite expensive. This is designed
 * to mostly be used by view accessors that are queried without any gas fees.
Developers should keep in mind that
 * this function has an unbounded cost, and using it as part of a state-changing
function may render the function
 * uncallable if the set grows to a point where copying to memory consumes too much
gas to fit in a block.
 */
function values(Bytes32Set storage set) internal view returns (bytes32[] memory) {
    bytes32[] memory store = _values(set._inner);
    bytes32[] memory result;

    /// @solidity memory-safe-assembly
    assembly {
        result := store
    }

    return result;
}
```

Figure 3.1: A snippet of the `OpenZeppelin EnumerableSet`'s detailing warning on gas

The `resolveDispute` function in the `PrivateERC20ResolutionModule` uses the `values` function to fetch the voters whose tokens should be transferred back to them:

```
/// @inheritdoc IPrivateERC20ResolutionModule
function resolveDispute(
  bytes32 _disputeId,
  IOracle.Request calldata _request,
  IOracle.Response calldata _response,
  IOracle.Dispute calldata _dispute
) external onlyOracle {
  [..]

  address[] memory __voters = _voters[_disputeId].values();

  [...]

  uint256 _length = __voters.length;
  for (uint256 _i; _i < _length;) {
    _params.votingToken.safeTransfer(__voters[_i],
_votersData[_disputeId][__voters[_i]].numOfVotes);
    unchecked {
      ++_i;
    }
  }
}
```

Figure 3.2: The `resolveDispute` function in `PrivateERC20ResolutionModule.sol`

The `EnumerableSet`'s `voters` function is also used in the `getVoters` function in the `ERC20ResolutionModule`:

```
function getVoters(bytes32 _disputeId) external view returns (address[] memory
__voters) {
  __voters = _voters[_disputeId].values();
}
```

Figure 3.3: The `getVoters` function in `ERC20ResolutionModule.sol`

Exploit Scenario

Alice, a user, wants to resolve a dispute. In doing so, the contracts copy a large array of voters from storage to memory. The `resolveDispute` call fails due to the size of the array, and the dispute cannot be resolved.

Recommendations

Short term, avoid using the `values` function in any capacity other than for view functions. Consider alternative ways to load the voters that do not involve the aforementioned function.

Long term, carefully review warnings provided by third-party dependencies to ensure that these do not cause integration issues.

4. Arbitrary request data can be provided to module functions in order to bypass validation	
Severity: **High**	Difficulty: **Low**
Type: Data Validation	Finding ID: TOB-WOND-4
Target: `prophet-modules/solidity/contracts/modules/resolution/*`	

Description

Because multiple modules do not validate the `requestId` against the provided `dispute` data, users can bypass module validation. This gives them the ability to bypass voting deadlines, bypass the reveal window, claim tokens before the dispute deadline, and manipulate dispute resolution by pledging arbitrary tokens.

The modules often use the provided request data to determine values to be used in the module functions' validation, as shown in figure 4.1:

```
function _pledge(
  IOracle.Request calldata _request,
  IOracle.Dispute calldata _dispute,
  uint256 _pledgeAmount,
  bool _pledgingFor
) internal {
  [...]
  RequestParameters memory _params =
decodeRequestData(_request.resolutionModuleData);

  uint256 _pledgingDeadline = _escalation.startTime + _params.timeUntilDeadline;

  if (block.timestamp >= _pledgingDeadline) revert
BondEscalationResolutionModule_PledgingPhaseOver();

  // Revert if the inequality timer has passed
  if (_inequalityData.time != 0 && block.timestamp >= _inequalityData.time +
_params.timeToBreakInequality) {
    revert BondEscalationResolutionModule_MustBeResolved();
  }

  _params.accountingExtension.pledge({
    _pledger: msg.sender,
    _requestId: _requestId,
    _disputeId: _disputeId,
    _token: _params.bondToken,
    _amount: _pledgeAmount
  });
  [...]
}
```

Figure 4.1: The `_pledge` function in `BondEscalationResolutionModule.sol`

However, some of the functions do not validate that the `_request` matches the `requestId` defined in the `_dispute`. This enables anyone to provide an arbitrary request in order to manipulate the system.

This issue is present in the following components:

- `BondEscalationResolutionModule`
- `BondEscalationModule`
- `PrivateERC20ResolutionModule`
- `ERC20ResolutionModule`

Exploit Scenario

Alice creates a request for pricing data and submits it to the `Oracle` contract. Eve submits a malicious response and her response is disputed and escalated. However, Eve deploys a ERC20 token for which she holds the entire token supply and calls the `pledgeAgainstDispute` function, setting the `bondToken` to her own token. This enables her to manipulate the result of the dispute.

Recommendations

Short term, ensure that the `requestId` is computed and compared against the `requestId` included in the `dispute`.

Long term, improve your testing suite by testing for common adversarial situations, such as providing an incorrect combination of inputs to the contract functions.

5. Ineffective access control can lead to user funds being stolen	
Severity: **High**	Difficulty: **Low**
Type: Access Controls	Finding ID: TOB-WOND-5
Target: `prophet-modules/solidity/contracts/extensions/{BondEscalationAccounting.sol, AccountingExtension.sol}`	

Description

The `onlyAllowedModule` modifier used as access control in the `BondEscalationAccounting` contract can be easily bypassed by creating an arbitrary request in the `Oracle` contract. This allows a malicious user to steal the token balance of any user for any token used inside the system.

The `BondEscalationAccounting` module allows users to deposit and pledge funds to be used for bond escalation. Several of the contract's functions use the `onlyAllowedModule` modifier to prevent arbitrary users from calling the contract functions and updating important state variables. The `AccountingExtension` uses the same modifier to provide access control to its functions:

```
modifier onlyAllowedModule(bytes32 _requestId) {
  if (!ORACLE.allowedModule(_requestId, msg.sender)) revert
AccountingExtension_UnauthorizedModule();
  _;
}
```

Figure 5.1: The `onlyAllowedModule` modifier in `AccountingExtension.sol`

The `allowedModule` function of the `Oracle` contract will look up if the address is included in the request modules and returns a Boolean. However, since anyone can create any arbitrary request in the `Oracle` contract, this means that the modifier can be easily bypassed.

A malicious user could exploit this issue to pledge to a dispute on behalf of someone else using any arbitrary token and amount, settle a bond escalation in order to freeze or steal assets, bond a user's tokens against their will in the `AccountingExtension`, trigger an underflow in the `claimEscalationReward` function, or release a pledge for any dispute to steal assets.

Exploit Scenario

A dispute is escalated and has gathered 100,000 USDC in value, pledged by various users. Eve notices the incorrect access control and creates a request that has her own address as an allowed module. She calls the `releasePledge` function and steals all of the tokens from this contract by bypassing the `onlyAllowedModule` access control. She repeats this action for any dispute that has a non-zero value of pledges in order to drain the system.

Recommendations

Short term, set the address of the modules that are expected to call functions from the `BondEscalationAccounting` and `AccountingExtension` contracts when deploying the contracts, and replace the `onlyAllowedModule` modifier with an access control that ensures that only one specific contract can call each function.

Long term, retroactively create an access control and data validation specification for the system. Use this specification to guide the further development of your testing suite and evaluate how access control can be modified to reduce the attack surface of the system. Consider redesigning the system so that each oracle has only one set of modules to reduce the attack surface of the system and to simplify access controls.

6. RootVerificationModule response can contain arbitrary data

Severity: **Low**	Difficulty: **Low**
Type: Data Validation	Finding ID: TOB-WOND-6
Target: `prophet-modules/solidity/contracts/modules/dispute/RootVerificationModule.sol`	

Description

The `RootVerificationModule` insufficiently validates the length of the `response` data, allowing users to append any arbitrary data after the first 32 bytes of the response. This could be used to deliver malformed data to the intended target of the fulfilled request, potentially manipulating the target contract.

The `RootVerificationModule` allows responses to be atomically verified by checking them against the calculated root of the Merkle tree provided in the request:

```
function disputeResponse(
  IOracle.Request calldata _request,
  IOracle.Response calldata _response,
  IOracle.Dispute calldata _dispute
) external onlyOracle {
  RequestParameters memory _params = decodeRequestData(_request.disputeModuleData);

  bytes32 _correctRoot = _params.treeVerifier.calculateRoot(_params.treeData,
_params.leavesToInsert);
  _correctRoots[_response.requestId] = _correctRoot;

  IOracle.DisputeStatus _status =
    abi.decode(_response.response, (bytes32)) != _correctRoot ?
IOracle.DisputeStatus.Won : IOracle.DisputeStatus.Lost;
  [...]
}
```

Figure 6.1: The `disputeResponse` function in `RootVerificationModule.sol`

If the response data matches the calculated root of the Merkle tree, this response is considered valid. However, the response data can contain any arbitrary data after the 32nd byte and still be considered valid. This is because `abi.decode` will truncate the response to 32 bytes.

Exploit Scenario

Alice creates a request that uses the `RootVerificationModule` as the dispute module and her own contract as the target for the finalized request. Eve calculates the root of the Merkle tree and adds it to the first 32 bytes of the response; however, she appends some arbitrary data after the 32nd byte. Eve's response is considered valid and is finalized, triggering a callback in the finality module and setting an important state variable in Alice's contract to an incorrect value.

Recommendations

Short term, check that `_response.response` is equal to 32 bytes in order to avoid arbitrary data being passed to the request target.

Long term, evaluate the expected bounds of all function parameters and ensure that they are explicitly enforced.

7. Executing the resolveDispute and revealVote functions in the same block can lead to loss of funds	
Severity: **Medium**	Difficulty: **Medium**
Type: Denial of Service	Finding ID: TOB-WOND-7
Target: `prophet-modules/solidity/contracts/modules/resolution/PrivateERC20ResolutionModule.sol`	

Description

If the `revealVote` function is executed after the `resolveDispute` function, provided this is done in the same block, the user that called `revealVote` will permanently lose access to their assets.

The `PrivateERC20ResolutionModule` allows users to vote on the validity of the dispute by submitting a hidden commitment that contains the information relevant to the vote, and later revealing this vote by calling the `commitVote` function. A vote can be revealed only if the `block.timestamp` is less than or equal to the end time of the reveal phase, as shown in figure 7.1:

```
function revealVote(
  IOracle.Request calldata _request,
  IOracle.Dispute calldata _dispute,
  uint256 _numberOfVotes,
  bytes32 _salt
) public {
  [...]
(uint256 _revealStartTime, uint256 _revealEndTime) = (
  _escalation.startTime + _params.committingTimeWindow,
  _escalation.startTime + _params.committingTimeWindow + _params.revealingTimeWindow
);

  if (block.timestamp <= _revealStartTime) revert
PrivateERC20ResolutionModule_OnGoingCommittingPhase();
  if (block.timestamp > _revealEndTime) revert
PrivateERC20ResolutionModule_RevealingPhaseOver();

  [...]

  _params.votingToken.safeTransferFrom(msg.sender, address(this), _numberOfVotes);

  emit VoteRevealed(msg.sender, _disputeId, _numberOfVotes);
}
```

Figure 7.1: The `revealVote` function in `PrivateERC20ResolutionModule.sol`

The `resolveDispute` function will update the dispute status and transfer the tokens back to all of the users that voted on this dispute. This function can be called only if the current `block.timestamp` is less than the end time of the reveal phase, as shown in figure 10.2:

```
function resolveDispute(...) external onlyOracle {
  [...]

  if (block.timestamp < _escalation.startTime + _params.committingTimeWindow) {
    revert PrivateERC20ResolutionModule_OnGoingCommittingPhase();
  }
  if (block.timestamp < _escalation.startTime + _params.committingTimeWindow +
_params.revealingTimeWindow) {
    revert PrivateERC20ResolutionModule_OnGoingRevealingPhase();
  }

  [...]

  uint256 _length = __voters.length;
  for (uint256 _i; _i < _length;) {
    _params.votingToken.safeTransfer(__voters[_i],
_votersData[_disputeId][__voters[_i]].numOfVotes);
    unchecked {
      ++_i;
    }
  }
}
```

Figure 7.2: The `resolveDispute` function in `PrivateERC20ResolutionModule.sol`

However, both functions can be called if the `block.timestamp` is exactly equal to the end of the reveal phase. This means that a user's call to the `revealVote` function could be front-run with a call to the `resolveDispute` function, permanently locking up their assets and influencing the result of the dispute.

Exploit Scenario

Alice commits 1,000 tokens to a dispute but chooses not to reveal her vote until the last moment. She sees that the vote has not reached quorum and decides to reveal her vote at the end of the reveal period. Eve notices this and front-runs her transaction with a call to the `resolveDispute` function. The dispute is marked as lost, but Alice's transaction still goes through, permanently locking the tokens she used to vote.

Recommendations

Short term, update the validation of the `resolveDispute` function so that it reverts if `block.timestamp` is less than or equal to the end of the reveal period.

Long term, improve the testing suite by adding tests for similar edge cases. Consider using stateful smart contract fuzzing with Echidna, Medusa, or Foundry to more easily discover these kinds of issues.

8. PrivateERC20ResolutionModule functions will always revert

Severity: **Medium**	Difficulty: **Low**
Type: Denial of Service	Finding ID: TOB-WOND-8
Target: `prophet-modules/solidity/contracts/modules/resolution/PrivateERC20ResolutionModule.sol`	

Description

The `commitVote` and `resolveDispute` functions will always revert, preventing disputes from being resolved.

The `commitVote` and `resolveDispute` functions of the `PrivateERC20ResolutionModule` contract check the dispute status to ensure that only an escalated dispute can be voted on or resolved. However, this check is incorrectly implemented in both functions, as shown in figure 8.1:

```
function commitVote(IOracle.Request calldata _request, IOracle.Dispute calldata
_dispute, bytes32 _commitment) public {
  bytes32 _disputeId = _getId(_dispute);
  if (ORACLE.createdAt(_disputeId) == 0) revert
PrivateERC20ResolutionModule_NonExistentDispute();
  if (ORACLE.disputeStatus(_disputeId) != IOracle.DisputeStatus.None) {
    revert PrivateERC20ResolutionModule_AlreadyResolved();
  }
  [...]
}
function resolveDispute(
  bytes32 _disputeId,
  IOracle.Request calldata _request,
  IOracle.Response calldata _response,
  IOracle.Dispute calldata _dispute
) external onlyOracle {
  if (ORACLE.createdAt(_disputeId) == 0) revert
PrivateERC20ResolutionModule_NonExistentDispute();
  if (ORACLE.disputeStatus(_disputeId) != IOracle.DisputeStatus.None) {
    revert PrivateERC20ResolutionModule_AlreadyResolved();
  }
  [...]
}
```

Figure 8.1: The `commitVote` and `resolveDispute` function in `PrivateERC20ResolutionModule.sol`

Since the status of a dispute whose resolution process has been started will be `Escalated` and not `None`, these functions will always revert.

Exploit Scenario

Alice creates a request that uses the `PrivateERC20ResolutionModule` contract as a resolution module. Bob proposes a response that Eve disputes and escalates. Due to the incorrect checks, users are unable to vote on the resolution of this dispute, permanently locking Bob's and Eve's bonded assets.

Recommendations

Short term, update the validation in the `commitVote` and `resolveDispute` functions to:
`ORACLE.disputeStatus(_disputeId) != IOracle.DisputeStatus.Escalated`

Long term, improve the testing suite by reducing the use of mock contracts and ensuring that the contracts are tested in a way that is as close as possible to how they are going to be used when deployed to production.

9. Dispute pledges for a NoResolution dispute can be stolen by anyone

Severity: **High**	Difficulty: **Low**
Type: Denial of Service	Finding ID: TOB-WOND-9
Target: `prophet-modules/solidity/contracts/extensions/BondEscalationAccounting.sol`	

Description

If a dispute status has been marked as `NoResolution`, anyone can steal all of the pledged tokens for this dispute.

Once an escalated dispute has been resolved, the `BondEscalationAccounting` contract allows users to withdraw their bonded tokens by calling the `claimEscalationReward` function. This function will check how many pledges the user has made and determine the amount of tokens to be assigned to this user's balance, as shown in figure 9.1:

```
function claimEscalationReward(bytes32 _disputeId, address _pledger) external {
  EscalationResult memory _result = escalationResults[_disputeId];
  if (_result.token == IERC20(address(0))) revert
BondEscalationAccounting_NoEscalationResult();
  bytes32 _requestId = _result.requestId;
  if (pledgerClaimed[_requestId][_pledger]) revert
BondEscalationAccounting_AlreadyClaimed();

  IOracle.DisputeStatus _status = ORACLE.disputeStatus(_disputeId);
  uint256 _amountPerPledger = _result.amountPerPledger;
  uint256 _numberOfPledges;

  if (_status == IOracle.DisputeStatus.NoResolution) {
    _numberOfPledges = 1;
  } else {
    _numberOfPledges = _status == IOracle.DisputeStatus.Won
      ? _result.bondEscalationModule.pledgesForDispute(_requestId, _pledger)
      : _result.bondEscalationModule.pledgesAgainstDispute(_requestId, _pledger);
  }

  IERC20 _token = _result.token;
  uint256 _claimAmount = _amountPerPledger * _numberOfPledges;

  pledgerClaimed[_requestId][_pledger] = true;
  balanceOf[_pledger][_token] += _claimAmount;

  unchecked {
    pledges[_disputeId][_token] -= _claimAmount;
  }
```

Figure 9.1: The `claimEscalationReward` function in `BondEscalationAccounting.sol`

However, in the highlighted line of the above figure, we can see that if the dispute status is `NoResolution`, anyone can withdraw one pledge, even if they have not pledged to this dispute. This allows any user to withdraw the full amount of pledges by calling this function with an arbitrary amount of different addresses.

Exploit Scenario
A dispute is escalated, and 1,000 tokens are pledged to the dispute. After some time passes, the dispute status is updated to `NoResolution`, and users are able to withdraw their pledge, 100 tokens per account. Eve creates 10 different accounts and calls the `claimEscalationReward` function for each one, stealing the entire pledge balance of this dispute.

Recommendations
Short term, replace the highlighted line of figure 9.1 with a call to the `pledgesForDispute` and `pledgesAgainstDispute` functions to dynamically determine the amount of pledges that are owed to a pledger.

Long term, improve the testing suite and ensure that it has full coverage over all of the contracts, statements, and branches.

10. Disputer’s escalation pledge may become locked	
Severity: **Medium**	Difficulty: **Medium**
Type: Data Validation	Finding ID: TOB-WOND-10
Target: `prophet-modules/solidity/contracts/extensions/BondEscalationAccounting.sol`	

Description

When settling a bond escalation, the `BondEscalation` contract does not call the `AccountingExtension` contract’s `onSettleBondEscalation` function unless at least one user has pledged against the dispute. If no users pledge against the dispute, any user who has pledged for the dispute will not be able to recover their pledge.

Figure 10.1 shows the logic handling dispute settlement in the case that the dispute escalation process does not result in a tie.

```
if (_pledgesAgainstDispute > 0) {
  uint256 _amountToPay = _won
    ? _params.bondSize
      + FixedPointMathLib.mulDivDown(_pledgesAgainstDispute, _params.bondSize,
_pledgesForDispute)
    : _params.bondSize
      + FixedPointMathLib.mulDivDown(_pledgesForDispute, _params.bondSize,
_pledgesAgainstDispute);

  _params.accountingExtension.onSettleBondEscalation({
    _requestId: _dispute.requestId,
    _disputeId: _escalation.disputeId,
    _token: _params.bondToken,
    _amountPerPledger: _amountToPay,
    _winningPledgersLength: _won ? _pledgesForDispute : _pledgesAgainstDispute
  });
}
```

Figure 10.1: An excerpt of the `onDisputeStatusChange` function of the `BondEscalationModule` contract

Exploit Scenario

Alice submits an incorrect response to a request. Bob disputes the response by posting a bond, then further escalates the dispute by submitting a pledge. Alice realizes that her response was wrong and does not pledge against the dispute. After the bond escalation period expires, the `BondEscalationModule` contract releases Bob’s bond but does not call the accounting extension's `onSettleBondEscalation`. Bob is then unable to successfully call `claimEscalationReward` to recover his pledge.

Recommendations

Short term, change the conditional to execute the `onSettleBondEscalation` logic when either of `_pledgesAgainstDispute` or `_pledgesForDispute` is nonzero.

Long term, add unit tests exercising all edge cases and conditionals in the contract logic.

11. Use of low-level calls can compromise access control	
Severity: **High**	Difficulty: **Low**
Type: Access Controls	Finding ID: TOB-WOND-11
Target: `prophet-modules/solidity/contracts`	

Description

The `CallbackModule`, `MultipleCallbacksModule`, and `CircuitResolverModule` contracts perform low-level calls with user-specified input data and target addresses. These contracts are trusted modules with substantial permissions in the overall protocol. Malicious users can call functions guarded with the `onlyAllowedModule` modifier in the `AccountingExtension` and `BondEscalationAccounting` contracts, potentially stealing user funds. They can also call functions in the `Oracle` contract such as `updateDisputeStatus`, leading to stolen funds or other invalid behavior.

```
/// @inheritdoc ICallbackModule
function finalizeRequest(
  IOracle.Request calldata _request,
  IOracle.Response calldata _response,
  address _finalizer
) external override(Module, ICallbackModule) onlyOracle {
  RequestParameters memory _params = decodeRequestData(_request.finalityModuleData);
  _params.target.call(_params.data);
  emit Callback(_response.requestId, _params.target, _params.data);
  emit RequestFinalized(_response.requestId, _response, _finalizer);
}
```

Figure 11.1: Unrestricted low-level call in `CallbackModule`

Exploit Scenario

A stablecoin protocol uses the Prophet system to fetch centralized exchange price data via the `HttpRequestModule`. It uses the `BondEscalationModule` for disputes and `MultipleCallbacksModule` for finalization. The protocol pre-fills the `Request` struct with appropriate data but allows end-users to specify a callback target to be notified when the result is finalized.

Mallory, a malicious user, submits a request that includes the address of `BondEscalationModule` as the callback target. She sets the callback data to match an ABI-encoded call to the pay function, transferring another user's bonded funds to herself.

Recommendations

Short term, replace the arbitrary low-level callbacks with a specific callback interface.

Long term, do not allow calls from privileged contracts to user-chosen endpoints.

Detailed Findings: Prophet Core

12. The wrong user can be charged for a resolved dispute	
Severity: **Medium**	Difficulty: **Low**
Type: Data Validation	Finding ID: TOB-WOND-12
Target: `prophet-core/solidity/contracts/Oracle.sol`	

Description

Due to missing validation on the `proposer` address included in the `dispute` data, the wrong user can be charged for a resolved dispute instead of the actual proposer of the response.

When a user submits a dispute for a particular response, the dispute will contain the `disputer` address, the address of the response `proposer`, the `responseId`, and the `requestId`:

```
struct Dispute {
  address disputer;
  address proposer;
  bytes32 responseId;
  bytes32 requestId;
}
```

Figure 12.1: The `Dispute` struct in `IOracle.sol`

While the `proposer` address is intended to be the address of the user that proposed the response, this is never validated in the `disputeResponse` function of the `Oracle` contract. If the disputer were to win the dispute, the address that will be charged the bond amount will be the proposer defined in the dispute struct:

```
} else if (_status == IOracle.DisputeStatus.Won) {
  // Disputer won, we pay the disputer and release their bond
  _params.accountingExtension.pay({
    _requestId: _dispute.requestId,
    _payer: _dispute.proposer,
    _receiver: _dispute.disputer,
    _token: _params.bondToken,
    _amount: _params.bondSize
  });
  [...]
```

Figure 12.2: A snipped from the `onDisputeStatusChange` function in `BondedDisputeModule.sol`

Due to this, a user is able to provide an arbitrary proposer address for a valid dispute in order to grief other users, causing them to lose their bond amount.

Exploit Scenario

Alice creates a request through the `Oracle` contract. Eve submits an incorrect response and disputes her own response, setting the response `proposer` address to Alice's address. Once the dispute is won, Alice will be charged the dispute bond instead of Eve; this will prevent her request from being finalized due to an insufficient balance.

Recommendations

Short term, ensure that the `proposer` address of a dispute is equal to the `proposer` of the response that the dispute is targeting.

Long term, improve your testing suite by testing for common adversarial situations such as providing an incorrect combination of inputs to the contract functions.

13. Insufficient validation of request/response existence could have unintended side effects	
Severity: **Undetermined**	Difficulty: **Low**
Type: Data Validation	Finding ID: TOB-WOND-13
Target: `prophet-core/solidity/contracts/Oracle.sol`	

Description

Multiple functions in the `Oracle` contract insufficiently validate if the request or response exists.

The `Oracle` contract allows users to create requests for arbitrary data and respond to other user's requests. The contract implements several functions that enable creating a request, submitting a response, disputing a response, escalating or resolving a dispute, and finalizing the request.

However, several functions insufficiently validate the existence of requests and responses:

- The `proposeResponse` function does not validate that the request exists, allowing users to propose responses to non-existent requests.
- The `disputeResponse` function does not validate that the response exists, allowing users to dispute a non-existent response.
- The `finalize` function does not validate that the request exists, potentially allowing users to finalize a non-existent request (depending on the modules included in the request).

Recommendations

Short term, ensure that all functions that assume a request, response, or dispute exists explicitly check that the `createdAt` mapping value for the `requestId`, `responseId`, or `disputeId` is not equal to zero.

Long term, improve the testing suite by adding additional negative unit tests to ensure that functions cannot be called out of order or with invalid input parameters. Evaluate the system for transaction ordering risks.

14. Externally owned accounts may be unable to submit requests during periods of high usage

Severity: **Low**	Difficulty: **Medium**
Type: Denial of Service	Finding ID: TOB-WOND-14
Target: `prophet-core/solidity/contracts/Oracle.sol`	

Description

Users are required to include a nonce in each request, which must correspond to a global request counter for the `Oracle` contract. When many requests are included in a single block, users interacting with the `Oracle` contract from an externally owned account may be unable to predict the correct nonce ahead of time and find that all of their request transactions are reverted.

Figure 14.1 shows the check that would fail if a user is unable to predict the correct nonce at the time of transaction signing.

```
function _createRequest(Request calldata _request, bytes32 _ipfsHash) internal
returns (bytes32 _requestId) {
  uint256 _requestNonce = totalRequestCount++;

  if (_requestNonce != _request.nonce || msg.sender != _request.requester) revert
Oracle_InvalidRequestBody();
```

Figure 14.1: Users must predict `totalRequestCount` before sending the transaction

If multiple users submit requests using the `totalRequestCount` value in the current state of the contract, then at most one of those users will have their transactions succeed.

Users interacting with the `Oracle` contract via a smart contract call will not experience this issue, since they can fetch the current count and then submit a request without preemption.

Exploit Scenario

Alice wants to make a request and uses a local DApp interface to interact with the `Oracle` smart contract. The DApp front end fetches the latest `totalRequestCount` value and includes it in a `Request` struct. Alice sends the transaction, which is then included in a block. However, earlier in the same block, Bob also makes an unrelated request. Alice's transaction then fails, consuming her transaction fee, without successfully submitting her request. Alice may need to try many times before the contract accepts her request.

Recommendations

Short term, allow users to submit requests with `Request.nonce` set to zero and overwrite the value with the current `totalRequestCount` value before further processing.

Long term, add tests exercising the case where multiple EOAs send interleaved transactions to the contracts.

15. A user’s balance can be bonded against their will

Severity: **High**	Difficulty: **Low**
Type: Denial of Service	Finding ID: TOB-WOND-15
Target: `prophet-core/solidity/contracts/Oracle.sol`	

Description

A user that has approved the response module to handle their tokens can have all of their tokens bonded against their will.

When proposing a response to a request, a user needs to approve the response module to handle their deposited token balance. This is done so that the response module is able to bond the user’s token for that request. Additionally, the `proposeResponse` function of the `Oracle` contract allows the dispute module of the request to provide an arbitrary proposer address whose tokens will be bonded. This is shown in the highlighted line of figure 15.1:

```
function proposeResponse(
  Request calldata _request,
  Response calldata _response
) external returns (bytes32 _responseId) {
  _responseId = _validateResponse(_request, _response);

  // The caller must be the proposer, unless the response is coming from a dispute
module
  if (msg.sender != _response.proposer && msg.sender !=
address(_request.disputeModule)) {
    revert Oracle_InvalidResponseBody();
  }

  // Can't propose the same response twice
  if (createdAt[_responseId] != 0) {
    revert Oracle_InvalidResponseBody();
  }

  if (finalizedAt[_response.requestId] != 0) {
    revert Oracle_AlreadyFinalized(_response.requestId);
  }

  _participants[_response.requestId] =
abi.encodePacked(_participants[_response.requestId], _response.proposer);
  IResponseModule(_request.responseModule).propose(_request, _response, msg.sender);
  _responseIds[_response.requestId] =
abi.encodePacked(_responseIds[_response.requestId], _responseId);
  createdAt[_responseId] = uint128(block.number);
  emit ResponseProposed(_response.requestId, _responseId, _response, block.number);
}
```

Figure 15.1: The `proposeResponse` function in `Oracle.sol`

However, anyone can bypass this validation by creating a malicious request that uses the same response module as the targeted user but sets the dispute module to their own address (or any address in their control). This means that any user is able to bond another user's tokens.

Exploit Scenario

Alice deposits 1,000 tokens into the accounting module of the request for which she intends to provide a response and approves the resolution module to handle her balance. Eve creates a malicious request with her own address to the dispute module and submits a response on behalf of Alice, bonding her tokens against her will.

Recommendations

Short term, consider enforcing that a single `Oracle` contract can serve only requests that have a specific and allowlisted set of modules included.

Long term, retroactively create an access control and data validation specification for the system. Use this specification to guide the further development of your testing suite and evaluate how access control can be modified to reduce the attack surface of the system. Consider redesigning the system so that each oracle has only one set of modules to reduce the attack surface of the system and simplify access control.

16. Lack of domain separation between request, response, and dispute IDs

Severity: **Informational**	Difficulty: **Medium**
Type: Cryptography	Finding ID: TOB-WOND-16
Target: `prophet-core/solidity/contracts/Oracle.sol`	

Description

Requests, responses, and disputes are uniquely identified in the protocol by identifiers computed as the Keccak256 hash of ABI-encoded structs. Due to a lack of domain separation, objects of two different types could theoretically serialize to the same ABI-encoded value and thus share a single ID.

Due to the specific ordering of fields in the `Request`, `Response`, and `Dispute` structs, we could not identify a specific scenario in which an adversary could force two different objects to share an ID. However, this is contingent on the exact order of fields in each struct, and the issue could easily become exploitable with even seemingly innocuous changes to the type definitions.

A collision between IDs of different types could lead to unexpected results when performing an existence check using the `_createdAt` array, which commingles IDs of all three types. For example, the `Oracle` contract's `finalize` function can pass an attacker-chosen `Response` object to the modules, so long as the `_response.requestID` value is zero.

```
function finalize(IOracle.Request calldata _request, IOracle.Response calldata
_response) external {
  bytes32 _requestId;
  bytes32 _responseId;

  // Finalizing without a response (by passing a Response with `requestId` == 0x0)
  if (_response.requestId == bytes32(0)) {
    _requestId = _finalizeWithoutResponse(_request);
  } else {
    (_requestId, _responseId) = _finalizeWithResponse(_request, _response);
  }

...

  IRequestModule(_request.requestModule).finalizeRequest(_request, _response,
msg.sender);

  emit OracleRequestFinalized(_requestId, _responseId, msg.sender, block.number);
}
```

Figure 16.1: The `finalize` function does not validate `_response` when `_response.requestID` is zero

The example module implementations then check whether the `_response` object is valid by checking the `createdAt` time rather than the `_response.requestID` value:

```
function finalizeRequest(
  IOracle.Request calldata _request,
  IOracle.Response calldata _response,
  address _finalizer
) external override(IHttpRequestModule, Module) onlyOracle {
  RequestParameters memory _params = decodeRequestData(_request.requestModuleData);

  if (ORACLE.createdAt(_getId(_response)) != 0) {
    _params.accountingExtension.pay({
```

Figure 16.2: Modules such as `HttpRequestModule` use `ORACLE.createdAt` to determine response validity

Thus, by creating a `Dispute` object with a colliding ID, an attacker could convince modules that an arbitrary nonexistent response object was finalized.

Exploit Scenario

A Prophet developer refactors the smart contracts and rearranges the `Dispute` struct to have the request and response IDs before the proposer and disputer addresses. An attacker uses a malicious dispute module to create a dispute with ABI encoding that could also be interpreted as the encoding of a response. The attacker calls `finalizeRequest` with a carefully crafted `_response` object that has `requestID` set to zero and an ID that matches the previously inserted dispute. The `Oracle` contract will interpret this as a nonexistent response, but the module contracts will look up the ID in the `createdAt` table and believe it to be a correct, validated response. The module contracts then act as if the attacker-chosen response has been finalized, causing loss of funds or inaccurate oracle results.

Recommendations

Short term, use a typed hashing format like EIP-712 to compute identifiers. Pass an all-zero `Response` object to modules during finalization without a valid response, rather than forwarding unvalidated data.

Long term, consider separating the `createdAt` mappings into type-specific mappings. Avoid passing any unvalidated data from the `Oracle` contract to `Module` contracts. Consider separating the module `finalizeRequest` interface into separate functions depending on whether or not a valid request is being finalized.

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
Cryptography and Key Management	The safe use of cryptographic primitives and functions, along with the presence of robust mechanisms for key generation and distribution
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
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
Transaction Ordering	The system's resistance to transaction-ordering attacks

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. Token Integration Checklist

The following checklist provides recommendations for interactions with arbitrary tokens. Every unchecked item should be justified and its associated risks understood. For an up-to-date version of the checklist, see `crytic/building-secure-contracts`.

For convenience, all Slither utilities can be run directly on a token address, such as the following:

```
slither-check-erc 0xdac17f958d2ee523a2206206994597c13d831ec7 TetherToken --erc erc20
slither-check-erc 0x06012c8cf97BEaD5deAe237070F9587f8E7A266d KittyCore --erc erc721
```

To follow this checklist, use the following output from Slither for the token:

```
slither-check-erc [target] [contractName] [optional: --erc ERC_NUMBER]
slither [target] --print human-summary
slither [target] --print contract-summary
slither-prop . --contract ContractName # requires configuration, and use of Echidna
and Manticore
```

General Considerations

- ❏ **The contract has a security review.** Avoid interacting with contracts that have not undergone a security review. Check the length of the assessment (i.e., the level of effort), the reputation of the security firm, and the number and severity of the findings.

- ❏ **You have contacted the developers.** You may need to alert their team to an incident. Look for appropriate contacts on `blockchain-security-contacts`.

- ❏ **The developers have a security mailing list for critical announcements.** Their team should advise users when critical issues are found or when upgrades occur.

Contract Composition

- ❏ **The contract avoids unnecessary complexity.** The token should be a simple contract; a token with complex code requires a higher standard of review. Use Slither's `human-summary` printer to identify complex code.

- ❏ **The contract uses `SafeMath` or Solidity 0.8.0+.** Contracts that do not use `SafeMath` require a higher standard of review. Inspect the contract by hand for `SafeMath`/Solidity 0.8.0+ usage.

- ❏ **The contract has only a few non-token-related functions.** Non-token-related functions increase the likelihood of an issue in the contract. Use Slither's `contract-summary` printer to broadly review the code used in the contract.

- ❏ **The token has only one address.** Tokens with multiple entry points for balance updates can break internal bookkeeping based on the address (e.g., `balances[token_address][msg.sender]` may not reflect the actual balance).

Owner Privileges

- ❏ **The token is not upgradeable.** Upgradeable contracts may change their rules over time. Use Slither's `human-summary` printer to determine whether the contract is upgradeable.

- ❏ **The owner has limited minting capabilities.** Malicious or compromised owners can misuse minting capabilities. Use Slither's `human-summary` printer to review minting capabilities, and consider manually reviewing the code.

- ❏ **The token is not pausable.** Malicious or compromised owners can trap contracts relying on pausable tokens. Identify pausable code by hand.

- ❏ **The owner cannot denylist the contract.** Malicious or compromised owners can trap contracts relying on tokens with a denylist. Identify denylisting features by hand.

- ❏ **The team behind the token is known and can be held responsible for misuse.** Contracts with anonymous development teams or teams that reside in legal shelters require a higher standard of review.

ERC-20 Tokens

ERC-20 Conformity Checks

Slither includes a utility, `slither-check-erc`, that reviews the conformance of a token to many related ERC standards. Use `slither-check-erc` to review the following:

- ❏ **`Transfer` and `transferFrom` return a Boolean.** Several tokens do not return a Boolean on these functions. As a result, their calls in the contract might fail.

- ❏ **The `name`, `decimals`, and `symbol` functions are present if used.** These functions are optional in the ERC-20 standard and may not be present.

- ❏ **`Decimals` returns a `uint8`.** Several tokens incorrectly return a `uint256`. In such cases, ensure that the value returned is less than 255.

- ❏ **The token mitigates the known ERC-20 race condition.** The ERC-20 standard has a known ERC-20 race condition that must be mitigated to prevent attackers from stealing tokens.

Slither includes a utility, `slither-prop`, that generates unit tests and security properties that can discover many common ERC flaws. Use `slither-prop` to review the following:

- ❏ **The contract passes all unit tests and security properties from `slither-prop`.** Run the generated unit tests and then check the properties with Echidna and Manticore.

Risks of ERC-20 Extensions

The behavior of certain contracts may differ from the original ERC specification. Conduct a manual review of the following conditions:

- ❏ **The token is not an ERC-777 token and has no external function call in `transfer` or `transferFrom`.** External calls in the transfer functions can lead to reentrancies.

- ❏ **`Transfer` and `transferFrom` should not take a fee.** Deflationary tokens can lead to unexpected behavior.

- ❏ **Potential interest earned from the token is accounted for.** Some tokens distribute interest to token holders. This interest may be trapped in the contract if not accounted for.

Token Scarcity

Reviews of token scarcity issues must be executed manually. Check for the following conditions:

- ❏ **The supply is owned by more than a few users.** If a few users own most of the tokens, they can influence operations based on the tokens' repartition.

- ❏ **The total supply is sufficient.** Tokens with a low total supply can be easily manipulated.

- ❏ **The tokens are in more than a few exchanges.** If all the tokens are in one exchange, a compromise of the exchange could compromise the contract relying on the token.

- ❏ **Users understand the risks associated with a large amount of funds or flash loans.** Contracts relying on the token balance must account for attackers with a large amount of funds or attacks executed through flash loans.

- ❏ **The token does not allow flash minting.** Flash minting can lead to substantial swings in the balance and the total supply, which necessitate strict and comprehensive overflow checks in the operation of the token.

ERC-721 Tokens

ERC-721 Conformity Checks

The behavior of certain contracts may differ from the original ERC specification. Conduct a manual review of the following conditions:

- ❏ **Transfers of tokens to the `0x0` address revert.** Several tokens allow transfers to `0x0` and consider tokens transferred to that address to have been burned; however, the ERC-721 standard requires that such transfers revert.
- ❏ **`safeTransferFrom` functions are implemented with the correct signature.** Several token contracts do not implement these functions. A transfer of NFTs to one of these contracts can result in a loss of assets.
- ❏ **The `name`, `decimals`, and `symbol` functions are present if used.** These functions are optional in the ERC-721 standard and may not be present.
- ❏ **If it is used, `decimals` returns a `uint8(0)`.** Other values are invalid.
- ❏ **The `name` and `symbol` functions can return an empty string.** This behavior is allowed by the standard.
- ❏ **The `ownerOf` function reverts if the `tokenID` is invalid or is set to a token that has already been burned.** The function cannot return `0x0`. This behavior is required by the standard, but it is not always properly implemented.
- ❏ **A transfer of an NFT clears its approvals.** This is required by the standard.
- ❏ **The `tokenID` of an NFT cannot be changed during its lifetime.** This is required by the standard.

Common Risks of the ERC-721 Standard

To mitigate the risks associated with ERC-721 contracts, conduct a manual review of the following conditions:

- ❏ **The `onERC721Received` callback is accounted for.** External calls in the transfer functions can lead to reentrancies, especially when the callback is not explicit (e.g., in `safeMint` calls).
- ❏ **When an NFT is minted, it is safely transferred to a smart contract.** If there is a minting function, it should behave like `safeTransferFrom` and properly handle the minting of new tokens to a smart contract. This will prevent a loss of assets.
- ❏ **The burning of a token clears its approvals.** If there is a burning function, it should clear the token's previous approvals.

D. Code Quality Recommendations

The following recommendations are not associated with specific vulnerabilities. However, implementing them can enhance the code's readability and may prevent the introduction of vulnerabilities in the future.

- The `PrivateERC20ResolutionModule` and `ERC20ResolutionModule` contracts use the numbers 1 and 0 to indicate if quorum has been reached. This can be replaced with a Boolean.
- The check that ensures the `block.timestamp` is not less than the end of the commit phase in the `resolveDispute` function of the `PrivateERC20ResolutionModule` is unnecessary since this check is included in the check after it. It can be removed.
- Multiple mappings are used to contain information about a request in the `Oracle` contract. This could be simplified by having a single mapping that points to a struct that contains all the necessary information.

E. Specification Guidelines

This section provides generally accepted best practices and guidance for how to write design specifications.

A good design specification serves three purposes:

1. **It helps development teams detect bugs/inconsistencies in a proposed system architecture before any code is written.** Codebases that were written without adequate specifications are often littered with snippets of code that have lost their relevance as the system's design has evolved. Without a specification, it is exceedingly challenging to detect such code snippets and remove them without extensive validation testing.

2. **It reduces the likelihood that bugs will be introduced in implementations of the system**. In systems without a specification, engineers must divide their attention between designing the system and implementing it in code. In projects requiring multiple engineers, engineers may make assumptions about how another engineer's component works, creating an opportunity for bugs to be introduced.

3. **It improves the maintainability of system implementations and reduces the likelihood that future code changes will introduce bugs.** Without an adequate specification, new developers need to spend time "on-ramping," where they explore the code and understand how it works. This process is highly error-prone and can lead to incorrect assumptions and the introduction of bugs.

Low-level designs may also be used by test engineers to create property-based fuzz tests and by auditors to reduce the amount of time needed to audit a specific protocol component.

Specification Construction

A good specification must describe system components in enough detail that an engineer unfamiliar with the project can use the specification to implement those components. The level of detail required to achieve this can vary from project to project, but generally, a low-level specification will include the following details, at minimum:

- Details about how each system component (e.g., a contract or plugin) interacts with and relies on other components

- The actors and agents that participate in the system, the way they interact with the system, and their permissions, roles, authorization mechanisms, and expected known-good flows

- The expected failure conditions the system may encounter and the way those failures are mitigated, including failures that are automatically mitigated
- Specification details for each function that the system will implement, which should include the following:
    - A description of the purpose of the function and its intended use
    - A description of the function's inputs and the various validations that are performed against each input
    - Any specific restrictions on the function's inputs that are not validated and not obvious
    - Any interactions between the function and other system components
    - The function's various failure modes, such as failure modes for queries to a module or accounting extension
    - Any authentication/authorization required by the function
    - Any function-level assumptions that depend on other components behaving in a specific way

In addition, specifications should use standardized RFC-2119 language as much as possible. This language pushes specification authors toward a writing style that is both detailed and easy to understand. One relevant example is the ERC-4626 specification, which uses RFC-2119 language and provides enough constraints on implementers so that a vault client for a single implementation may be used interchangeably with other implementations.

Interaction Specification Example

An interaction specification is used to describe how the components of the system depend on each other. It includes a description of the other components that the system interacts with, the nature of those interactions, expected behavior or dependencies, and access relationships.

Note that a diagram can often be a helpful aid for modeling component interactions, but it should not be used to substitute a textual description of the component's interactions. Part of the goal of a specification is to help derive a list of properties that can be explicitly tested, and deriving properties from a diagram is much more challenging and error-prone than from a textual specification.

Here is an example of an interaction specification. The sections in brackets are intentionally left out for simplicity.

`BondEscalationAccounting` interacts with the following contracts:

- *`BondEscalationModule`: `BondEscalationAccounting` queries this contract to determine the amount of pledges that an address has made for a particular dispute, if this dispute does not have the `NoResolution` status. This is used in the `claimEscalationReward` function to determine the amount of rewards a user is due.*
- *`Oracle`: `BondEscalationAccounting` queries this contract to determine the dispute status of a dispute ID provided as an input to the `claimEscalationReward` function. This is necessary since the `Oracle` contract is the single source of truth for the status of a dispute.*

The following contracts interact with `BondEscalationAccounting`:

- *[Example contract]*

The following actors interact with `BondEscalationAccounting`:

- *[Example actor]*

Function Specification Example

Here is an example of a specification for the `claimEscalationReward` function.

The `claimEscalationReward(bytes32 _disputeId, address _pledger)` function can be called by anyone.

*If the provided `_disputeId` has not been settled, this function **must** revert.*

*If the provided `_pledger` address has already claimed their tokens for a particular `_disputeId`, this function **must** revert.*

*If the provided `_pledger` address has not pledged any tokens to the provided dispute, the function **must** revert.*

*If the dispute was won, the users that pledged their tokens in favor of the dispute **must** be able to receive their reward. The users that pledged their tokens against the dispute **must not** be able to receive a reward.*

If the dispute was lost, the above condition is opposite.

*If a dispute was not resolved, any user that has pledged their tokens either in favor or against the dispute **must** be able to withdraw the exact amount that they have pledged.*

Another complementary technique for defining a function specification, which can be especially useful for defining test cases, is the branching tree technique (proposed by Paul

Berg), which is a tree-like structure based on all of the execution paths, the contract state or function arguments that lead to each path, and the end result of each path.

Figure E.1 shows an example of a branching tree specification for the `claimEscalationReward` function:

```
claimEscalationReward(bytes32 _disputeId, address _pledger)
├── when the _disputeId has not been resolved
│   └── it must revert
└── when the _disputeId has been resolved
    ├── when the dispute status is Won
    │   ├── when the _pledger has voted in favor of the dispute
    │   │    ├── it should increase _pledger balance by their number of pledges
multiplied by the amount per pledge
    │   │    ├── it should reduce the total pledge balance for the dispute by the same
amount
    │   │    └── it should mark the _pledger as claimed
    │   ├── when the _pledger has votes against the dispute
    │   │    ├── it should increase _pledger balance by their number of pledges
multiplied by the amount per pledge
    │   │    ├── it should reduce the total pledge balance for the dispute by the same
amount
    │   │    └── it should mark the _pledger as claimed
    │   └── when the _pledger has not voted
    │        └── it must revert
    ├── when the dispute status is Loss
    ├── when the dispute status is NoResolution
    │   ├── when the pledger has voted either in favor or against the dispute
    │   │    ├── it should increase _pledger balance by their number of pledges
multiplied by the amount per pledge
    │   │    ├── it should reduce the total pledge balance for the dispute by the same
amount
    │   │    └── it should mark the _pledger as claimed
    │   └── when the pledger has not voted in the dispute
    │        └── it must revert
...
```

Figure E.1: An example branching tree specification for the `claimEscalationReward` function

This type of specification can be useful when developing unit tests, as it makes it easy to identify the execution paths, conditions, and edge cases that need to be tested.

F. Mutation Testing

This appendix outlines how we conducted mutation testing and highlights some of the most actionable results.

At a high level, mutation tests make several changes to each line of a target file and rerun the test suite against each change. Changes that result in test failures indicate adequate test coverage, while changes that do not cause tests to fail indicate gaps in test coverage. Mutation allows auditors to focus their review on areas of the codebase that are most likely to contain latent bugs, and it allows developers to identify and add missing tests.

We used an experimental new mutation tool, `slither-mutate`, to conduct our mutation testing campaign. This tool is custom-made for Solidity and features higher performance and fewer false positives than existing tools such as `universalmutator`.

```
python3 -m pip install slither-analyzer
```

Figure F.1: Installing Slither using pip

The mutation campaign was run against the in-scope smart contracts using the following commands:

```
slither-mutate . --test-cmd='yarn test' --test-dir='solidity/test'
--ignore-dirs='scripts,lib,test,node_modules' --timeout 100
```

Figure F.2: A Bash script that runs a mutation testing campaign against each Solidity file in the `solidity/contracts` directory

Consider the following notes about the above commands:

- On a consumer-grade laptop, the overall runtime of the mutation testing campaign for the module contract is approximately 17 hours, while the campaign for the core contracts is approximately one hour.
- The `--test-cmd` flags specify the command to run to assess mutant validity. A `--fail-fast` or `--bail` flag will automatically be added to test commands to improve runtime.

An abbreviated, illustrative example of a mutation test output file is shown in figure F.3.

```
INFO:Slither-Mutate:[MIA] Line 76: '_response.requestId != _requestId' ==>
'!(_response.requestId != _requestId)' --> UNCAUGHT
INFO:Slither-Mutate:[MIA] Line 97: '_response.requestId != _requestId' ==> 'true'
--> UNCAUGHT
INFO:Slither-Mutate:[MIA] Line 97: '_response.requestId != _requestId' ==> 'false'
--> UNCAUGHT
```

```
INFO:Slither-Mutate:[MIA] Line 97: '_response.requestId != _requestId' ==>
'!(_response.requestId != _requestId)' --> UNCAUGHT
INFO:Slither-Mutate:[MVIE] Line 74: 'bytes32 _requestId = ' ==> 'bytes32 _requestId
' --> UNCAUGHT
INFO:Slither-Mutate:[MVIE] Line 92: 'bytes32 _requestId = ' ==> 'bytes32 _requestId
' --> UNCAUGHT
INFO:Slither-Mutate:[MVIE] Line 93: 'bytes32 _responseId = ' ==> 'bytes32
_responseId ' --> UNCAUGHT
INFO:Slither-Mutate:[ROR] Line 76: '_response.requestId != _requestId' ==>
'_response.requestId  <  _requestId' --> UNCAUGHT
INFO:Slither-Mutate:[ROR] Line 76: '_response.requestId != _requestId' ==>
'_response.requestId  >  _requestId' --> UNCAUGHT
INFO:Slither-Mutate:[ROR] Line 76: '_response.requestId != _requestId' ==>
'_response.requestId  <=  _requestId' --> UNCAUGHT
INFO:Slither-Mutate:[ROR] Line 76: '_response.requestId != _requestId' ==>
'_response.requestId  >=  _requestId' --> UNCAUGHT
INFO:Slither-Mutate:[ROR] Line 76: '_response.requestId != _requestId' ==>
'_response.requestId  ==  _requestId' --> UNCAUGHT
INFO:Slither-Mutate:[ROR] Line 97: '_response.requestId != _requestId' ==>
'_response.requestId  <  _requestId' --> UNCAUGHT
INFO:Slither-Mutate:[ROR] Line 97: '_response.requestId != _requestId' ==>
'_response.requestId  >  _requestId' --> UNCAUGHT
INFO:Slither-Mutate:[ROR] Line 97: '_response.requestId != _requestId' ==>
'_response.requestId  <=  _requestId' --> UNCAUGHT
INFO:Slither-Mutate:[ROR] Line 97: '_response.requestId != _requestId' ==>
'_response.requestId  >=  _requestId' --> UNCAUGHT
INFO:Slither-Mutate:[ROR] Line 97: '_response.requestId != _requestId' ==>
'_response.requestId  ==  _requestId' --> UNCAUGHT
INFO:Slither-Mutate:Done mutating Module.
INFO:Slither-Mutate:Revert mutants: 5 uncaught of 5 (100.0%)
INFO:Slither-Mutate:Comment mutants: 1 uncaught of 2 (50.0%)
INFO:Slither-Mutate:Tweak mutants: 26 uncaught of 30 (86.66666666666667%)
```

Figure F.3: Abbreviated output from the mutation testing campaign assessing test coverage of the in-scope contracts

In summary, the mutation testing campaign has caught:

- 32 valid mutants in the `Module` contract,
- Six valid mutants in the `Oracle` contract,
- 25 valid mutants in the `BondEscalationAccounting` contract,
- and 10 valid mutants in the `AccountingExtension` contract.

The following is list of contracts that includes a subset of functions and features of each contract that lack sufficient tests:

- `Oracle` insufficiently tests:
    - That the caller is added to the `_participants` state variable when calling the `proposeResponse` and `disputeResponse` functions,

    - That the `createdAt` state variable is updated when calling the `disputeResponse` function,
    - That the `_matchBytes` function correctly breaks,
    - That the call to the request module is made when executing the `_createRequest` function,
    - That the `_validateDispute` function can revert if the dispute IDs do not match the request and response IDs.
- `Module`
    - This contract is untested in the prophet-core repository.
- `BondEscalationAccounting` insufficiently tests:
    - The user balance updates in the `pledge` function,
    - The reduction of the `pledged[disputeId][_token]` state variable in the `claimEscalationReward` function,
    - User balance updates and the reduction of the `pledged[disputeId][_token]` state variable in the `releasePledge` function,
    - Calling the `claimEscalationReward` function on a dispute with a status of `NoResolution`,
    - Calling the `releasePledge`, `pledge`,and `onSettleBondEscalation` functions with an amount that is not exactly equal to `pledged[disputeId][_token]`,
- `AccountingExtension` insufficiently tests:
    - The emission of the `Bonded` event,
    - Updates to the `bondedAmountOf` and `balanceOf` state variables in the `bond` functions,
    - Calling the bond functions with an amount that is not exactly equal to `balanceOf[_bonder][_token]`

Additionally, contracts for which no valid mutants were generated should be reviewed to ensure that the mock contracts and calls are not forcing the tests to pass even if impactful code changes are made.

We recommend that the Wonderland team review the existing tests and add additional verification that would catch the aforementioned types of mutations, as well as minimize the amount of mock contracts and calls used in the tests. Then, use a script similar to that provided in figure F.2 to rerun a mutation testing campaign to ensure that the added tests provide adequate coverage.

G. Fix Review Results

When undertaking a fix review, Trail of Bits reviews the fixes implemented for issues identified in the original report. This work involves a review of specific areas of the source code and system configuration, not comprehensive analysis of the system.

On August 22, 2024, Trail of Bits reviewed the fixes and mitigations implemented by the Wonderland team for the issues identified in this report. We reviewed each fix to determine its effectiveness in resolving the associated issue.

In summary, of the 16 issues described in this report, Wonderland has resolved 15 issues and has partially resolved one issue. For additional information, please see the Detailed Fix Review Results below.

ID	Title	Status
1	Lack of contract existence check on low-level calls	Resolved
2	Contracts are incompatible with ERC20 tokens with fee	Resolved
3	Dispute resolution can be blocked due to high gas costs	Partially Resolved
4	Arbitrary request data can be provided to module functions in order to bypass validation	Resolved
5	Ineffective access control can lead to user funds being stolen	Resolved
6	RootVerificationModule response can contain arbitrary data	Resolved
7	Executing the resolveDispute and revealVote functions in the same block can lead to loss of funds	Resolved
8	PrivateERC20ResolutionModule functions will always revert	Resolved
9	Dispute pledges for a NoResolution dispute can be stolen by anyone	Resolved
10	Disputer's escalation pledge may become locked	Resolved
11	Use of low-level calls can compromise access control	Resolved

12	The wrong user can be charged for a resolved dispute	Resolved
13	Insufficient validation of request/response existence could have unintended side effects	Resolved
14	Externally owned accounts may be unable to submit requests during periods of high usage	Resolved
15	A user's balance can be bonded against their will	Resolved
16	Lack of domain separation between request, response, and dispute IDs	Resolved

Detailed Fix Review Results

TOB-WOND-1: Lack of contract existence check on low-level calls
Resolved in PR 52 and PR 56. The contract existence check was added to the `validateParameters` functions, which are intended to be called off-chain. While this does not directly mitigate the issue, the callback interfaces introduced in PR 56 will lead to the addition of an implicit contract existence check before the calls are performed. As a result, calls to externally owned accounts will revert, mitigating this issue.

TOB-WOND-2: Contracts are incompatible with ERC20 tokens with fee
Resolved in PR 42. Additional documentation was added to the repository that explicitly states that fee-on-transfer tokens are not supported. A check was added to the `deposit` function of the `AccountingExtension` contract that ensures that the amount deposited is equal to the amount received.

TOB-WOND-3: Dispute resolution can be blocked due to high gas costs
Partially resolved in PR 50. The proposed fix does not use the `values` function of OpenZeppelin's `EnumerableSet` library; however, it still loops over an array of unbounded size. This could lead to a similar risk of denial of service due to high gas costs.

TOB-WOND-4: Arbitrary request data can be provided to module functions in order to bypass validation
Resolved in PR 51. The contracts outlined in this finding have been updated so that functions that do not have access controls always verify that the request, response, and dispute combinations are valid.

TOB-WOND-5: Ineffective access control can lead to user funds being stolen
Resolved in PR 62. Additional verification was added to validate that a request, response, or dispute pair is valid. Addresses that are authorized to call the `BondEscalationModule`

contract's functions are set in the constructor, and a onlyAuthorizedCaller modifier was added. This modifier ensures that only authorized addresses can call these functions.

TOB-WOND-6: RootVerificationModule response can contain arbitrary data
Resolved in PR 44. The disputeResponse function of the RootVerificationModule now checks that the length of the response is exactly 32 bytes before decoding it.

TOB-WOND-7: Executing the resolveDispute and revealVote functions in the same block can lead to loss of funds
Resolved in PR 45. The validation of the resolveDispute function was updated so that it reverts if block.timestamp is less than or equal to the end of the reveal period.

TOB-WOND-8: PrivateERC20ResolutionModule functions will always revert
Resolved in PR 46. The validation in the commitVote and resolveDispute functions of the PrivateERC20ResolutionModule contract was updated so that these functions can be called only for escalated disputes.

TOB-WOND-9: Dispute pledges for a NoResolution dispute can be stolen by anyone
Resolved in PR 47. The claimEscalationReward function of the BondEscalationAccounting contract has been updated so that if a dispute has the NoResolution status, the amount of pledges for a user are dynamically calculated. This prevents the dispute pledges from being stolen for NoResolution disputes.

TOB-WOND-10: Disputer's escalation pledge may become locked
Resolved in PR 48. The incorrect conditional has been updated to execute the onSettleBondEscalation logic when either _pledgesAgainstDispute or _pledgesForDispute is nonzero.

TOB-WOND-11: Use of low-level calls can compromise access control
Resolved in PR 56. The arbitrary low-level calls were replaced by the prophetVerify and prophetCallback callback interfaces, preventing the call target from being a module or the Oracle contract.

TOB-WOND-12: The wrong user can be charged for a resolved dispute
Resolved in PR 28. The dispute.proposer check was updated so that the field correctly points to the proposer of the response that is being disputed.

TOB-WOND-13: Insufficient validation of request/response existence could have unintended side effects
Resolved in PR 37. The request, response, and dispute ID calculation and validation functions have been moved to the ValidatorLib library contract. Multiple checks were added to the Oracle contract functions:

- The `proposeResponse` function now checks that the request exists, preventing the creation of responses for non-existent requests.
- The `disputeResponse` function now checks that the response exists, preventing a non-existent response from being disputed.
- The `escalateDispute`, `resolveDispute`, and `updateDisputeStatus` functions now check that the dispute exists.
- The `_finalizeWithoutResponse` function checks that the request exists, and the `_finalizeWithResponse` function checks that the response exists. This prevents non-existent requests or responses from being finalized.

TOB-WOND-14: Externally owned accounts may be unable to submit requests during periods of high usage
Resolved in PR 27. The `_createRequest` function now allows users to submit requests with `Request.nonce` set to zero, in which case it overwrites the value with the current `totalRequestCount` value before further processing.

TOB-WOND-15: A user's balance can be bonded against their will
Resolved in PR 54. The bond function of the `AccountingExtension` contract now checks that the sender and the module are approved to bond a user's tokens. This prevents the scenario described in the corresponding finding. However, the `Oracle` contract can still serve requests to any arbitrary module, which increases the attack surface of the system. Care should be taken to review the system for other risks related to use of arbitrary module contracts.

TOB-WOND-16: Lack of domain separation between request, response, and dispute IDs
Resolved in PR 33. The `createdAt` mapping was separated into three mappings, one each for requests, responses, and disputes. This prevents any collision between the request/response/dispute IDs from affecting the `Oracle` contract. However, the user-controlled `response` input is still passed to the module contracts even when a request is finalized without a response. This should be safe due to the condition that the `response.requestId` needs to be zero, but offloading security to the modules could create additional developer/security overhead.

H. Fix Review Status Categories

The following table describes the statuses used to indicate whether an issue has been sufficiently addressed.

Fix Status	
Status	**Description**
Undetermined	The status of the issue was not determined during this engagement.
Unresolved	The issue persists and has not been resolved.
Partially Resolved	The issue persists but has been partially resolved.
Resolved	The issue has been sufficiently resolved.