Pyth Data Association Entropy

Security Assessment

January 18, 2024

Prepared for:
Pyth Data Association

Prepared by: **Tjaden Hess and Elvis Skoždopolj**

About Trail of Bits

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

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

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

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

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

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

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

Notices and Remarks

Copyright and Distribution

Test Coverage Disclaimer

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

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

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

Table of Contents

Project Summary

Contact Information

The following project manager was associated with this project:

Jeff Braswell, Project Manager
jeff.braswell@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:

Tjaden Hess, Consultant
tjaden.hess@trailofbits.com

Elvis Skoždopolj, Consultant
elvis.skozdopolj@trailofbits.com

Project Timeline

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

Date	Event
November 30, 2023	Pre-project kickoff call
December 11, 2023	Status update meeting #1
December 18, 2023	Delivery of report draft
December 18, 2023	Report readout meeting
January 18, 2024	Delivery of comprehensive report

Executive Summary

Engagement Overview

Pyth Data Association engaged Trail of Bits to review the security of the `Entropy` and `Executor` smart contracts and the Fortuna web service. Pyth Entropy is a permissionless, two-party random number generator (RNG) protocol that uses a commitment-and-reveal scheme to generate random numbers. The Fortuna web service facilitates the process of registering as a provider, requesting a random number, and revealing the random number. The `Executor` contract allows the execution of arbitrary messages verified by the Wormhole cross-chain bridge.

A team of two consultants conducted the review from December 4 to December 15, 2023, for a total of four engineer-weeks of effort. Our testing efforts focused on determining whether third parties can influence or control the generated random number, bypass access control rules, or bypass message verification to execute arbitrary messages. We also examined the risks associated with integrating the Entropy system with higher-level protocols. With full access to source code and documentation, we performed static and dynamic testing of the codebase, using automated and manual processes.

Observations and Impact

The security of protocols integrating with the Pyth Entropy service depends heavily on a lack of bias in output values and the inability of any party to know the eventual output values in advance. We identified one serious implementation issue affecting the quality of randomness output (TOB-ENTR-4), which would allow a user to choose between two different values for the eventual output. Additionally, the protocol as a whole relies on non-collusion assumptions that may vary depending on the integrating application, making secure integration error-prone (TOB-ENTR-5).

Overall, the codebase is cleanly implemented, readable, and well documented but lacking in thorough automated testing, especially for the Fortuna Rust application (TOB-ENTR-11).

Recommendations

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

- **Remediate the findings disclosed in this report.** These findings should be addressed as part of a direct remediation or as part of any refactor that may occur when addressing other recommendations.

- **Provide integration examples.** Ensure that higher-level protocols can securely integrate with Fortuna by providing examples of a variety of use cases and clearly documenting all trust assumptions and selective-abort considerations.
- **Improve the testing suites.** Ensure that the testing suites have full coverage for all protocol components and features. Consider using advanced testing techniques such as fuzzing and mutation testing to help identify issues that might be difficult to find via manual analysis, and ensure the robustness of the testing suite.

Finding Severities and Categories

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

EXPOSURE ANALYSIS

Severity	*Count*
High	4
Medium	1
Low	3
Informational	4
Undetermined	0

CATEGORY BREAKDOWN

Category	*Count*
Configuration	2
Cryptography	1
Data Exposure	2
Data Validation	5
Denial of Service	1
Testing	1

Project Goals

The engagement was scoped to provide a security assessment of the Pyth Data Association `Entropy` and `Executor` smart contracts and the Fortuna web service. Specifically, we sought to answer the following non-exhaustive list of questions:

- Can a user have excess influence over the random number generation?
- Are the fees attributed correctly to the providers and the protocol?
- Can providers withdraw an excess amount of fees?
- Can the protocol withdraw the accrued protocol fees?
- Can users avoid paying the fee?
- Are the protocol actions robust against reentrancy?
- Can the Wormhole message verification be bypassed?
- Can randomness requests be reexecuted?
- Can users learn the contribution of providers before committing to their own contribution?
- Does the Fortuna service adequately validate requests and protect secrets?

Project Targets

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

Entropy core contracts

Repository	https://github.com/pyth-network/pyth-crosschain/tree/main/target_chains/ethereum/contracts/contracts/entropy
Version	`c592fd36cab3145e007df56280f4a94f0cb9fa62`
Type	Solidity
Platform	Ethereum

Executor contracts

Repository	https://github.com/pyth-network/pyth-crosschain/tree/main/target_chains/ethereum/contracts/contracts/executor
Version	`c592fd36cab3145e007df56280f4a94f0cb9fa62`
Type	Solidity
Platform	Ethereum

Fortuna

Repository	https://github.com/pyth-network/pyth-crosschain/tree/main/fortuna
Version	`c592fd36cab3145e007df56280f4a94f0cb9fa62`
Type	Rust
Platform	Native

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:

- **Smart contracts.** We performed manual review of the smart contracts and used Slither to statically analyze them. We reviewed the contracts for common Solidity flaws, such as missing contract existence checks on low-level calls, issues with access controls, susceptibility to reentrancy attacks, unimplemented functions, and issues related to upgradeable contracts, such as state variable storage slot collision and initialization front running. We investigated each of the system components:
    - **`Entropy` contract**. We reviewed the way entropy is requested and revealed in the system, investigating whether a request for entropy can be overwritten or influenced by other system users, whether an invalidated entropy request can be revealed, and whether the users or providers can have an outsized influence on the revealed random number through collusion or any system flaws. We also checked how fees are accrued and distributed inside the system, whether excess fees can be extracted, and whether parts of the fee can become stuck in the contracts.
    - **`Executor` contract**. We reviewed the way encoded Wormhole messages are verified and decoded to determine whether message verification can be bypassed. We checked the validation of messages and whether received native assets can be withdrawn from the contract. We very briefly reviewed the Wormhole contracts the system integrates with to determine whether additional security measures are needed in the `Executor` contract.
- **Fortuna Rust application.** We performed a manual review of the Fortuna source code, checking for vulnerabilities that would allow users to learn secret entropy values outside the window specified by the protocol. We investigated the handling and configuration of secret values, the construction of hash-chain commitments, and the API interfaces for interacting with the application. We also reviewed the application for denial-of-service issues and Rust best practices.

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 did not perform an in-depth review of the Wormhole contracts used to manage ownership of and authenticate messages for the `Executor` contract.

- We did not review any contracts for chains other than Ethereum.
- We did not review any part of the `pyth-crosschain` repository except those outlined in the Project Targets section.

Automated Testing

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

Test Harness Configuration

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

Tool	Description	Policy
Slither	A static analysis framework that can statically verify algebraic relationships between Solidity variables	Appendix C
Clippy	An open-source Rust linter used to catch common mistakes and unidiomatic Rust code	Appendix C
`cargo-llvm-cov`	A Cargo plugin for generating LLVM source–based code coverage	Appendix C
Necessist	A framework that runs tests with statements and method calls removed to help identify broken tests	Appendix C
Semgrep	A tool for running crowd-sourced and custom lints to detect known vulnerabilities and anti-patterns	Appendix C

Areas of Focus

Our automated testing and verification work focused on the following:

- Existence of common vulnerabilities in the code
- General code quality issues and unidiomatic code patterns
- General issues with dependency management and known vulnerable dependencies
- Issues with unit test coverage

Test Results

We ran several static analysis tools such as Clippy, Slither, and Necessist to identify potential code quality issues in the codebase. We ran `cargo-llvm-cov` to review the Fortuna unit test coverage.

Property	Tool	Result
Native assets cannot be locked in the contracts.	Slither	TOB-ENTR-1
Contract existence checks are performed on low-level calls.	Slither	TOB-ENTR-2
The block hash value is checked.	Slither	TOB-ENTR-4
The project adheres to Rust best practices by fixing code quality issues reported by linters such as Clippy.	Clippy	Appendix D
All components of the codebase have sufficient test coverage.	`cargo-llvm-cov`	TOB-ENTR-11
Removing statements or method calls from tests causes the tests to fail.	Necessist	Appendix D
The codebase does not contain known code anti-patterns.	Semgrep	Passed

Codebase Maturity Evaluation

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

Category	Summary	Result
Arithmetic	All system contracts use built-in, safe arithmetic operations, and no complex arithmetic is present in the in-scope contracts.	**Strong**
Auditing	Events are emitted from most critical state-changing functions, although we found two instances where additional event emission would improve the ease of monitoring the system. We are unaware of any off-chain monitoring system. The protocol would benefit from such a system so that the correct functioning of the protocol could be tracked and so that alerts could be configured in case of suspicious activity. Additionally, creating an incident response plan to define a recovery process in case of a compromise would improve the protocol's security posture (see appendix E).	**Moderate**
Authentication / Access Controls	The system implements only two access control roles, which can execute a limited number of privileged actions. Both the roles are intended to be instances of the `Executor` contract that are controlled by Pyth governance on Solana, but creating additional documentation that describes the nature of the roles and the circumstances under which the role actions should be executed would be beneficial. Additionally, both roles are updated using a single-step process, which can be error-prone (TOB-ENTR-3).	**Moderate**
Complexity Management	The system architecture is simple, with well-defined functions and components and minimal code duplication.	**Satisfactory**

Cryptography and Key Management	We did not find any issues with the use of cryptography in the Entropy system. However, the Fortuna provider application does not adequately protect secret data from accidental disclosure (TOB-ENTR-9).	**Moderate**
Decentralization	The `Entropy` contract is upgradeable by the contract owner, which is intended to be the `Executor` contract. All contract upgrades will be managed by Pyth governance, which was not in scope for this review, and executed through the `Executor` contract. This reduces the risk of the owner becoming compromised and executing malicious upgrades. However, users are unable to opt out of the upgrade, which could potentially update user or provider commitments. The `Executor` contract is non-upgradeable, and its state cannot be updated.	**Moderate**
Documentation	The codebase contains thorough documentation in the form of inline comments. However, no other form of documentation, such as system specifications, architecture diagrams, user stories, or specification of roles, is currently available. The developer- and user-facing documentation currently available is limited, providing only a high-level overview of the system and incomplete recommendations on patterns and best practices. Pyth Data Association could benefit from creating thorough documentation of the system and improving the developer- and user-facing documentation.	**Moderate**
Low-Level Manipulation	The system uses a limited amount of low-level assembly, which is sufficiently commented.	**Satisfactory**
Testing and Verification	The testing suite contains unit tests for normal protocol use and some known adversarial situations. The tests are run as part of the CI. However, some of the issues present in the report (TOB-ENTR-1) could have been caught using unit tests, indicating that coverage of the `Entropy` and `Executor` testing suite could be improved.	**Moderate**

	The Fortuna provider application contains only two tests for the API; this application would benefit from the creation of a robust testing suite covering all aspects of the application.	
Transaction Ordering	We did not identify any issues related to transaction reordering.	**Strong**

Summary of Findings

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

ID	Title	Type	Severity
1	Deposited assets cannot be withdrawn	Configuration	High
2	Lack of contract existence check on low-level call	Data Validation	Low
3	Lack of two-step process for critical operations	Data Validation	Medium
4	Users can influence the Entropy revealed result	Data Validation	High
5	Integrating protocols may be vulnerable to multiparty collusion attacks	Configuration	High
6	Lack of zero-value checks	Data Validation	Low
7	Entropy providers may reveal seed before request is finalized	Data Exposure	High
8	Fortuna entropy seed does not bind provider identity	Cryptography	Informational
9	Secrets appear in environment variables and command-line arguments	Data Exposure	Informational
10	Calls to the reveal function may succeed on inactive requests	Data Validation	Informational
11	Insufficient unit tests for Fortuna	Testing	Informational
12	Provider may earn fees without disclosing entropy	Denial of Service	Low

Detailed Findings

1. Deposited assets cannot be withdrawn

Severity: **High**	Difficulty: **Low**
Type: Configuration	Finding ID: TOB-ENTR-1
Target: `Entropy.sol`, `Executor.sol`	

Description

The `Entropy` and `Executor` contracts can receive native asset deposits but cannot withdraw them, permanently locking the funds in the contracts.

The `Entropy` contract allows users to request random numbers from a randomness provider by calling the `Entropy` contract's `request` function. To make a request, the user must pay a fee, which consists of the provider-defined fee and the Pyth protocol fee, as shown in the highlighted lines of figure 1.1.

```
function request(
    address provider,
    bytes32 userCommitment,
    bool useBlockHash
) public payable override returns (uint64 assignedSequenceNumber) {
    EntropyStructs.ProviderInfo storage providerInfo = _state.providers[
        provider
    ];
    if (_state.providers[provider].sequenceNumber == 0)
        revert EntropyErrors.NoSuchProvider();

    // Assign a sequence number to the request
    assignedSequenceNumber = providerInfo.sequenceNumber;
    if (assignedSequenceNumber >= providerInfo.endSequenceNumber)
        revert EntropyErrors.OutOfRandomness();
    providerInfo.sequenceNumber += 1;

    // Check that fees were paid and increment the pyth / provider balances.
    uint128 requiredFee = getFee(provider);
    if (msg.value < requiredFee) revert EntropyErrors.InsufficientFee();
    providerInfo.accruedFeesInWei += providerInfo.feeInWei;
    _state.accruedPythFeesInWei += (SafeCast.toUint128(msg.value) -
        providerInfo.feeInWei);
    [...]
}
```

Figure 1.1: The `request` function of `Entropy.sol`

While providers can withdraw their accrued fees by using the `Entropy` contract's `withdraw` function, the contract does not implement a way for the Pyth protocol fee to be withdrawn. This results in the protocol fee being permanently locked in the contract.

This issue is also present in the `Executor` contract, which is designed to execute arbitrary messages that were verified by the Wormhole cross-chain bridge. The bridge can receive native assets because it implements a `payable receive` function, as shown in figure 1.2.

```
/// @dev Called when `msg.value` is not zero and the call data is empty.
receive() external payable {}
```

Figure 1.2: The `payable receive` function of `Executor.sol`

However, the `Executor` contract's `execute` function cannot send value with the executed low-level call, as shown in figure 1.3.

```
function execute(
    bytes memory encodedVm
) public returns (bytes memory response) {
    IWormhole.VM memory vm = verifyGovernanceVM(encodedVm);

    GovernanceInstruction memory gi = parseGovernanceInstruction(
        vm.payload
    );

    if (gi.targetChainId != chainId && gi.targetChainId != 0)
        revert ExecutorErrors.InvalidGovernanceTarget();

    if (
        gi.action != ExecutorAction.Execute ||
        gi.executorAddress != address(this)
    ) revert ExecutorErrors.DeserializationError();

    bool success;
    (success, response) = address(gi.callAddress).call(gi.callData);
    [...]
}
```

Figure 1.3: The `execute` function of `Executor.sol`

No additional mechanism for withdrawing the value is implemented inside the contract, so any native assets deposited to the contract will be permanently locked, barring a contract upgrade.

Exploit Scenario

The `Entropy` system accrues 1 ether of Pyth protocol fees. Alice, a Pyth team member, goes to withdraw the fees but realizes she cannot because the functionality is missing from the contract. The Pyth team must upgrade the contract to add this functionality.

Recommendations

Short term, add a withdrawal function that allows an authorized user to withdraw the accrued Pyth protocol fee from the `Entropy` contract. Consider also adding a withdrawal function, or allowing value to be attached to the low-level call, in the `Executor` contract.

Long term, improve the coverage of the Wallet testing suite and ensure that all system components and functionality are thoroughly tested.

2. Lack of contract existence check on low-level call

Severity: **Low**	Difficulty: **High**
Type: Data Validation	Finding ID: TOB-ENTR-2
Target: `Executor.sol`	

Description

The `Executor` contract uses a low-level call on arbitrary receivers but does not implement a contract existence check. If the call receiver is set to an incorrect address, such as an externally owned account (EOA), the call will succeed and could cause the protocol team to incorrectly assume that the attempted action has been performed.

The `Executor` contract is intended to execute messages that were previously transmitted and verified using the Wormhole cross-chain bridge in order to perform governance-approved actions on the receiver chain. This is done by calling the `execute` function with the appropriate encoded message, as shown in figure 2.1.

```
function execute(
    bytes memory encodedVm
) public returns (bytes memory response) {
    IWormhole.VM memory vm = verifyGovernanceVM(encodedVm);

    GovernanceInstruction memory gi = parseGovernanceInstruction(
        vm.payload
    );

    if (gi.targetChainId != chainId && gi.targetChainId != 0)
        revert ExecutorErrors.InvalidGovernanceTarget();

    if (
        gi.action != ExecutorAction.Execute ||
        gi.executorAddress != address(this)
    ) revert ExecutorErrors.DeserializationError();

    bool success;
    (success, response) = address(gi.callAddress).call(gi.callData);

    // Check if the call was successful or not.
    if (!success) {
        // If there is return data, the delegate call reverted with a reason or a
custom error, which we bubble up.
        if (response.length > 0) {
            // The first word of response is the length, so when we call revert we
add 1 word (32 bytes)
```

```
            // to give the pointer to the beginning of the revert data and pass the
size as the second argument.
            assembly {
                let returndata_size := mload(response)
                revert(add(32, response), returndata_size)
            }
        } else {
            revert ExecutorErrors.ExecutionReverted();
        }
    }
}
```

Figure 2.1: The `execute` function of `Executor.sol`

However, if the `gi.callAddress` parameter is mistakenly set to an EOA, the call will always succeed. Because the call to `execute` does not revert, the protocol team may assume that important actions have been performed, even though no action has been executed.

Exploit Scenario

Alice, a Pyth team member, submits a cross-chain message through Wormhole to execute a time-sensitive transaction but inputs the wrong `gi.callAddress`. She calls `execute` with the message data, which passes because the call is made to an EOA. Alice believes the intended transaction was successful and only realizes her mistake after it is too late to resubmit the correct message.

Recommendations

Short term, implement a contract existence check before the low-level call to ensure that the call reverts if the receiver is an EOA.

Long term, carefully review the Solidity documentation, especially the “Warnings” section about using low-level call operations.

3. Lack of two-step process for critical operations

Severity: **Medium**	Difficulty: **High**
Type: Data Validation	Finding ID: TOB-ENTR-3
Target: `EntropyUpgradable.sol`	

Description

When called, the `transferOwnership` function immediately sets the contract owner to the provided address. The use of a single step to make such a critical change is error-prone; if the function is called with erroneous input, the results could be irrevocable or difficult to recover from.

The `EntropyUpgradeable` contract inherits ownership logic from the OpenZeppelin `OwnableUpgradeable` contract, which allows the current owner to transfer the contract ownership to another address using the `transferOwnership` function, as shown in figure 3.1.

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

Figure 3.1: The `transferOwnership` function of `OwnableUpgradeable.sol`

The system also defines an administrator address that can be updated by calling the `EntropyGovernance` contract's `setAdmin` function, as shown in figure 3.2.

```
function setAdmin(address newAdmin) external {
    _authoriseAdminAction();

    address oldAdmin = _state.admin;
    _state.admin = newAdmin;

    emit AdminSet(oldAdmin, newAdmin);
}
```

Figure 3.2: The `setAdmin` function of `EntropyGovernance.sol`

Both of these critical operations are done in a single step. If the functions are called with erroneous input, the Pyth team could lose the ability to upgrade the contract or set important system parameters.

Exploit Scenario

Alice invokes `transferOwnership` to change the contract owner but accidentally enters the wrong address. She permanently loses access to the contract.

Recommendations

Short term, implement a two-step process for all irrecoverable critical operations, such as by replacing the `OwnableUpgradeable` contract with the `Ownable2StepUpgradeable` contract. Consider splitting the `setAdmin` function into a `proposeAdmin` function, which proposes the new admin address, and an `acceptAdmin` function, which sets the new admin address and can be called only by the proposed admin. This will guarantee that the admin-setting party must be able call the contract from the proposed address before they are actually set as the new owner.

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

4. Users can influence the Entropy revealed result

Severity: **High**	Difficulty: **Low**
Type: Data Validation	Finding ID: TOB-ENTR-4
Target: `Entropy.sol`	

Description

When revealing a requested random number that includes a block hash, users can choose between two random numbers by executing the `reveal` function in the same block as the request, or after 256 blocks. This gives each user two chances to receive a favorable result on each request.

The Entropy system generates random numbers by using a commitment-and-reveal pattern, which requires the third-party integration to first request a number and then later reveals the random number. This can be done by performing the following steps:

1. A user first selects a number, which they must keep secret, and provides the hash of the number to the `Entropy` contract's `request` function, along with the address of the randomness provider and a flag that determines whether the block hash should be used when generating the result.
2. The user queries the provider off-chain to reveal their secret number so that the user can provide it in the next step.
3. The user calls the `Entropy` contract's `reveal` function and provides the address of the provider, the sequence number, and both secrets in order to obtain the resulting random number.

The `reveal` function concatenates the user secret, the provider secret, and the block hash to generate the random number, as shown in the highlighted lines of figures 4.1 and 4.2.

```
function reveal(
     address provider,
     uint64 sequenceNumber,
     bytes32 userRandomness,
     bytes32 providerRevelation
 ) public override returns (bytes32 randomNumber) {
     EntropyStructs.Request storage req = findRequest(
         provider,
         sequenceNumber
     );
```

```
    [...]

    bytes32 blockHash = bytes32(uint256(0));
    if (req.useBlockhash) {
        blockHash = blockhash(req.blockNumber);
    }

    randomNumber = combineRandomValues(
        userRandomness,
        providerRevelation,
        blockHash
    );

    [...]
}
```

Figure 4.1: The `reveal` function of `Entropy.sol`

```
function combineRandomValues(
    bytes32 userRandomness,
    bytes32 providerRandomness,
    bytes32 blockHash
) public pure override returns (bytes32 combinedRandomness) {
    combinedRandomness = keccak256(
        abi.encodePacked(userRandomness, providerRandomness, blockHash)
    );
}
```

Figure 4.2: The `combineRandomValues` function of `Entropy.sol`

However, if the request specifies that the block hash should be used to generate the random number, this gives the user one additional random number and allows them to select whichever one is more favorable for them. This is because the `blockhash` function returns zero if the `req.blockNumber` variable is equal to the current block number, or if it is not within the 256 most recent blocks.

Exploit Scenario

A gambling protocol integrates with the `Entropy` contract to generate random numbers for their draws. They specify that the block hash must be used to generate the number but allow users to choose when to reveal the number with the limitation that if a number is not revealed within one day of the draw, the protocol will automatically mark the user as having lost the draw.

Eve notices that she can receive two random numbers (doubling her chances of success) by selecting when to reveal, and she submits multiple requests. She simulates the result of the draw for each of the two random numbers she can generate for that draw and ultimately chooses the winning number.

Recommendations

Short term, add validation when using the block hash to ensure that the `reveal` function reverts if the returned block hash is zero.

Long term, carefully review the Solidity documentation and ensure that the risks of using the `blockhash` function, especially as they relate to randomness generation, are well documented and understood.

5. Integrating protocols may be vulnerable to multiparty collusion attacks	
Severity: **High**	Difficulty: **High**
Type: Configuration	Finding ID: TOB-ENTR-5
Target: `Entropy.sol`	

Description

Users and providers may be able to collude to extract value from systems integrating with the Entropy RNG protocol.

The Pyth `Entropy` contract works via a two-party commitment-and-reveal protocol. The final generated value is derived from two secret seed values generated by the user and the provider. Each party must submit a hiding commitment to their seed before they can learn the seed of the other party. When the `blockhash` option is enabled, the resulting value additionally incorporates the hash of the block containing the request.

When the `blockhash` option is not enabled, the user and provider in multiparty settings may collude to bias the output of the contract, potentially extracting value from third parties. These attacks are very difficult to detect, as they are outwardly identical to low-probability events occurring by random chance.

When the `blockhash` option is enabled, biasing the output requires participation from the user, the provider, and a block proposer.

Integrating protocols and their users may not be aware of the Entropy protocol's underlying trust assumptions. While there are some in-code comments describing the non-collusion assumption, there is little in the way of user-facing documentation about assumptions or code examples demonstrating secure integration in multiparty settings.

The following are some use-case scenarios in which higher-level protocols may be vulnerable to collusion attacks:

1. An NFT sale uses the Entropy service to determine attributes, and thus value, of newly minted NFTS. In this situation, the user is the buyer of the NFT and the provider is controlled by the NFT creator.

   The NFT creator may pose as a user and collude with the entropy provider to create biased outputs and mint themselves valuable, rare NFTs, secretly reducing the value available to honest buyers.

2. Another NFT sale uses a third-party entropy provider that is not controlled by the NFT creator.

   The third-party provider may purchase NFTs and use their knowledge of provider seeds to bias the output and receive high-value NFTs.

3. A single-winner lottery decentralized application (dapp) chooses a winner by paying out to the user whose entropy value is lowest.

   The entropy provider purchases a ticket and manipulates their entropy outcome to be lower than other players', depriving them of a fair chance at winning the lottery.

4. A single-winner lottery dapp chooses a winner by combining the entropy value of each user into a single value.

   A user can bias the output by choosing not to reveal their seed. If the user owns many tickets, the resulting bias may earn the user more in expectations than the cost of forfeiting one lottery ticket.

Exploit Scenario

Alice, a randomness provider, deploys a gambling contract on Ethereum that integrates with the Entropy system and sets herself as the randomness provider. The contract attracts 10 ether of value from other users. Alice additionally stakes enough Ether to become a validator and waits for a slot in which she is the block proposer.

Alice, aware of the provider's secret, generates several candidate blocks, each containing a request with a distinct user commitment. Knowing all three of the user secret, provider secret, and candidate block hash, Alice can easily choose the candidate block that maximizes her chances of winning the lottery. She uses this knowledge to win the gamble and the 10 ether of deposited value.

Recommendations

Short term, explicitly document the trust considerations and recommended architectures for common use cases. Provide code examples demonstrating multiuser systems and elaborate on the trust issues involved.

Long term, consider implementing an architecture that allows fewer concentrated trust assumptions, such as a threshold verifiable random function (VRF) or a zkSNARK-based verifiable delay function (VDF).

6. Lack of zero-value checks

Severity: **Low**	Difficulty: **Low**
Type: Data Validation	Finding ID: TOB-ENTR-6
Target: `Entropy.sol`, `EntropyGovernance.sol`, `Executor.sol`	

Description

Certain setter functions and constructors fail to validate incoming arguments, so callers of these functions, or the deployer of the contracts, could mistakenly set important state variables to a zero value, misconfiguring the system.

For example, the `Executor` contract constructor sets multiple parameters, including the `wormhole` address, which is the address of the `Wormhole` contract used to verify encoded messages.

```
constructor(
    address _wormhole,
    uint64 _lastExecutedSequence,
    uint16 _chainId,
    uint16 _ownerEmitterChainId,
    bytes32 _ownerEmitterAddress
) {
    wormhole = IWormhole(_wormhole);
    lastExecutedSequence = _lastExecutedSequence;
    chainId = _chainId;
    ownerEmitterChainId = _ownerEmitterChainId;
    ownerEmitterAddress = _ownerEmitterAddress;
}
```

Figure 6.1: The `constructor` of `Executor.sol`

If the `wormhole` address is set to a zero value, the contract will be unable to execute transactions and will need to be redeployed.

The lack of zero-value checks is prevalent in the following contracts and functions:

- `Executor.sol`
  - constructor
- `Entropy.sol`
  - `_initialize` function

- `EntropyGovernance.sol`
    - `setAdmin` function
    - `setPythFee` function
    - `setDefaultProvider` function

Exploit Scenario
Alice deploys the `Executor` contract with a misconfigured `wormhole` address. As a result, the contract is unable to function, requiring Alice to redeploy it and pay the deployment fee again.

Recommendations
Short term, add zero-value checks to all function and constructor arguments to ensure that callers cannot set incorrect values and misconfigure the system.

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

7. Entropy providers may reveal seed before request is finalized

Severity: **High**	Difficulty: **High**
Type: Data Exposure	Finding ID: TOB-ENTR-7
Target: `fortuna/src/api.rs`	

Description

The Fortuna entropy provider service determines a chain's finality based on the number of confirmations (i.e., the number of blocks that include a given transaction in a parent block). This measure of finality is adequate for blockchains with probabilistic finality (e.g., Ethereum proof-of-work [PoW], Avalanche) and chains with instant finality (e.g., Solana, Tendermint) but is not adequate for Ethereum proof-of-stake (PoS) or L2s based on it.

Ethereum's consensus exhibits eventual finality, more commonly referred to as a *finality gadget*. Consensus systems using finality gadgets allow new blocks to be created when only one-third of the validator set is active and online. These blocks are organized into batches called epochs. At the end of an epoch, the validator set votes to finalize all the blocks in the epoch. This finalization requires two-thirds of the validator set to be honest in order to succeed.

If a chain using a finality gadget is attacked, an attacker may be able to prevent the chain from finalizing new epochs. However, since the act of creating new blocks has much better liveness properties, the chain will continue to produce blocks regardless of whether the finality gadget has stalled. This means that it is possible for many epochs to pass and remain unfinalized, which creates the possibility of extraordinarily long block reorgs under worst-case conditions.

Because a reorg may allow a user to change their entropy commitment, it is imperative that the randomness provider not reveal their seed for the corresponding sequence number until the user's request is final. Confirmation delays have no bearing on whether a block in Ethereum PoS is final.

Exploit Scenario

An attacker discovers a bug in an Ethereum consensus client and obtains the ability to prevent the chain from finalizing for several epochs. The attacker also controls several validators that have upcoming block proposal slots.

The attacker then submits a `request` transaction that includes some entropy commitment. Once the transaction is included in a block, the attacker begins an attack to prevent the chain from finalizing. This attack must continue for longer than the period indicated by the

`reveal_delay_blocks` variable. Once the correct number of blocks has passed, the attacker requests the corresponding entropy seed from the randomness provider. The attacker uses the provider's entropy to brute force a desirable new user seed. They then force a reorg of the chain to a point before the initial `request` transaction and submit a new transaction with the commitment corresponding to the alternative seed.

After this, the attacker can generate a new entropy result that is arbitrarily biased in their favor, potentially causing loss of funds in the higher-level protocol.

Recommendations

Short term, validate API requests for entropy revelation by checking the presence of a corresponding request at the most recent finalized block. Figure 7.1 provides a suggestion of how to do this using the `ethers-rs` library:

```
let r = self
.get_request(provider_address, sequence_number)
.block(ethers::core::types::BlockNumber::Finalized)
.call()
.await?;
```

Figure 7.1: Use of finalized block to prevent reorg attacks (`fortuna/src/chain/ethereum.rs:190–193`)

Long term, research the finality conditions of each chain to be supported, and ensure that providers do not reveal seeds before user commitments are finalized.

References

- The Engineer's Guide To Finality
- Post-Mortem Report: Ethereum Mainnet Finality (05/11/2023)

8. Fortuna entropy seed does not bind provider identity

Severity: **Informational**	Difficulty: **High**
Type: Cryptography	Finding ID: TOB-ENTR-8
Target: `fortuna/src/state.rs`	

Description

The Fortuna entropy provider service derives the base seed for individual hash chains by combining a master secret with a random nonce, the hash chain length, and the blockchain ID. The use of the blockchain ID prevents a provider from unintentionally revealing seeds on one chain that might be later used on another chain. However, the derivation does not include the provider's address. Figure 8.1 shows the derivation of a hash chain seed from a master secret:

```
pub fn from_config(
    secret: &str,
    chain_id: &ChainId,
    random: &[u8; 32],
    chain_length: u64,
) -> Result<Self> {
    let mut input: Vec<u8> = vec![];
    input.extend_from_slice(&hex::decode(secret)?);
    input.extend_from_slice(&chain_id.as_bytes());
    input.extend_from_slice(random);

    let secret: [u8; 32] = Keccak256::digest(input).into();
    Ok(Self::new(secret, chain_length.try_into()?))
}
```

Figure 8.1: The hash chain derivation does not include the provider's address. (pyth-crosschain/fortuna/src/state.rs#35–48)

Because the master seed is passed into the Fortuna application separately from the provider private key, it is possible that a misconfigured application use could result in the same entropy secret being used with two different provider accounts.

Under most circumstances, reuse of a master secret would not lead to an exploit due to the use of a random nonce to instantiate individual hash chains; new hash chains generated under the second provider address would be unlinkable to entropy generated by the first.

However, upon startup, the Fortuna service retrieves the random nonce from on-chain commitment metadata, as shown in figure 8.2. Thus, if an honest Fortuna instance is run using a constant entropy seed and a potentially untrusted provider address, the owner of the untrusted address could duplicate a commitment from the honest provider and extract entropy values that could be used to bias the outcome of Entropy queries from the trusted provider.

```
let provider_info = contract.get_provider_info(opts.provider).call().await?;
...
let metadata =
    bincode::deserialize::<CommitmentMetadata>(&provider_info.commitment_metadata)?;

let hash_chain = PebbleHashChain::from_config(
    &opts.randomness.secret,
    &chain_id,
    &metadata.seed,
    metadata.chain_length,
)?;
```

Figure 8.2: Fortuna fetches a random nonce from the blockchain. (pyth-crosschain/fortuna/src/command/run.rs#54–71)

This issue requires an unusual misconfiguration and use case for the Fortuna service, but due to the use of environment variables to store the secret values (TOB-ENTR-9), misconfigurations of this variety are far from impossible.

Exploit Scenario

An honest hosting service offers to run the Fortuna application on behalf of entropy providers, who supply their blockchain private key to the hosting service to enable the service to run on the provider's behalf. The hosting service fails to properly regenerate the `FORTUNA_SECRET` environment variable when changing providers.

Alice, an honest provider, uses the hosting service to generate entropy for an NFT mint. The hosting service generates a hash chain and uploads a commitment for Alice's entropy. Mallory, a malicious user, observes Alice's commitment and registers an identical commitment under Mallory's own provider. She then supplies her private key to the hosting provider and submits several requests for entropy. Because the entropy is not bound to the provider account, the revealed values will be the same as for future requests to Alice's service.

Mallory then uses her knowledge of Alice's entropy seeds to generate biased values during the NFT mint, obtaining valuable NFTs unfairly.

Recommendations

Short term, include the provider address as a component in the derivation of base seeds for new hash chains.

Long term, consider also including the initial sequence number and any other information necessary to uniquely identify a hash chain.

9. Secrets appear in environment variables and command-line arguments

Severity: **Informational**	Difficulty: **High**
Type: Data Exposure	Finding ID: TOB-ENTR-9
Target: `fortuna/src/config`	

Description

The Fortuna application expects the provider private key and master entropy seed to be passed through the command-line interface (CLI) arguments `--private-key` and `--secret`, or via the environment variables `PRIVATE_KEY` and `FORTUNA_SECRET`. As a result, an attacker with access to a different unprivileged user on the machine may, under some configurations, be able to learn the secrets being used by the Fortuna service.

On many Linux machines, any user can read environment variables and command-line flags via the `procfs` filesystem. Additionally, secrets may be unintentionally logged to the `.bash_history` file, which could have undesirably broad read permissions.

Exploit Scenario

An attacker exploits another process on the same host as Fortuna and uses it to learn the entropy seed or provider secret key from the list of processes.

Recommendations

Short term, have the code read secrets from a file on disk or prompt for secrets at runtime.

Long term, ensure that no secrets are passed via command-line arguments. Consider encrypting local secrets at rest or documenting integration with a secret storage solution such as HashiCorp Vault or Square Keywhiz.

10. Calls to the reveal function may succeed on inactive requests

Severity: **Informational**	Difficulty: **Low**
Type: Data Validation	Finding ID: TOB-ENTR-10
Target: `Entropy.sol`	

Description

Revealing inactive `Entropy` requests may succeed due to missing validation, allowing requests with a sequence number of zero to be revealed.

The `Entropy` contract uses a commitment-and-reveal scheme to generate random numbers. Users can request a random number from any registered randomness provider by using the `request` function, and once the provider has shared their secret with them, they can reveal the resulting random number by calling the `reveal` function. This function finds the corresponding request based on the provider address and the request sequence number, performs validation on it, and then finally clears the request so that it cannot be reexecuted, as shown in figure 10.1.

```
function reveal(
    address provider,
    uint64 sequenceNumber,
    bytes32 userRandomness,
    bytes32 providerRevelation
) public override returns (bytes32 randomNumber) {
    EntropyStructs.Request storage req = findRequest(
        provider,
        sequenceNumber
    );
    // Check that there is a request for the given provider / sequence number.
    if (req.provider != provider || req.sequenceNumber != sequenceNumber)
        revert EntropyErrors.NoSuchRequest();

    if (req.requester != msg.sender) revert EntropyErrors.Unauthorized();

    [...]

    clearRequest(provider, sequenceNumber);

    [...]
}
```

Figure 10.1: The `reveal` function of `Entropy.sol`

The `findRequest` function generates the request keys—the short key representing an index in the `_state.requests` fixed-size array and the key representing a key in the `_state.requestsOverflow` mapping—and matches one of the requests, as shown in figure 10.2.

```
function findRequest(
    address provider,
    uint64 sequenceNumber
) internal view returns (EntropyStructs.Request storage req) {
    (bytes32 key, uint8 shortKey) = requestKey(provider, sequenceNumber);

    req = _state.requests[shortKey];
    if (req.provider == provider && req.sequenceNumber == sequenceNumber) {
        return req;
    } else {
        req = _state.requestsOverflow[key];
    }
}
```

Figure 10.2: The `findRequest` function of `Entropy.sol`

When clearing a request at the end of a reveal, the `clearRequest` function either deletes the request from the `_state.requestsOverflow` mapping or invalidates it by setting the sequence number of the request to zero, as shown in figure 10.3.

```
function clearRequest(address provider, uint64 sequenceNumber) internal {
    (bytes32 key, uint8 shortKey) = requestKey(provider, sequenceNumber);

    EntropyStructs.Request storage req = _state.requests[shortKey];
    if (req.provider == provider && req.sequenceNumber == sequenceNumber) {
        req.sequenceNumber = 0;
    } else {
        delete _state.requestsOverflow[key];
    }
}
```

Figure 10.3: The `clearRequest` function of `Entropy.sol`

However, while a request with a sequence number of zero is considered inactive, the `reveal` function does not revert if sequence number zero is passed as an input to the function. Since the `_state.requests` fixed-sized array has a size of 32 elements, it is reasonable to assume that two different sequence numbers could result in the same short key, allowing inactive requests to be revealed.

Recommendations

Short term, add validation to the `reveal` function to ensure that it reverts if the sequence number is zero.

Long term, use advanced testing techniques such as fuzzing to more easily discover issues such as this. Defining a property that an inactive request cannot be revealed and using Echidna to test the property could help discover this issue.

11. Insufficient unit tests for Fortuna

Severity: **Informational**	Difficulty: **Not Applicable**
Type: Testing	Finding ID: TOB-ENTR-11
Target: Fortuna application	

Description

The Fortuna Rust codebase currently contains only two tests, both exercising aspects of the entropy revelation component of Fortuna. However, Fortuna comprises several other functionalities, such as provider registration and entropy request generation. The overall statement test coverage, as measured by `cargo llvm-cov`, is less than 20%.

Additionally, the testing codebase currently relies on mocked versions of the Entropy contract ABI. Failing to routinely exercise the `ethers-rs` components and realistic chain behavior may lead to undiscovered bugs and unexpected failures.

Recommendations

Short term, add unit and integration tests for all subcommands of the Fortuna service.

Long term, add integration tests exercising the full set of Fortuna commands against a local blockchain testnet node in a faithful simulation of real-world use.

12. Provider may earn fees without disclosing entropy

Severity: **Low**	Difficulty: **Low**
Type: Denial of Service	Finding ID: TOB-ENTR-12
Target: `Entropy.sol`	

Description

Entropy providers may set a configurable fee as compensation for providing the entropy. However, they may collect the fee without disclosing their entropy to the user and thus collect fees simply by registering once and then going offline.

Users must pay the provider fee as part of submitting a request, at which point the funds are immediately available for withdrawal by the provider, as demonstrated in figure 12.1.

```
function request(
    address provider,
    bytes32 userCommitment,
    bool useBlockHash
) public payable override returns (uint64 assignedSequenceNumber) {
...

    // Check that fees were paid and increment the pyth / provider balances.
    uint128 requiredFee = getFee(provider);
    if (msg.value < requiredFee) revert EntropyErrors.InsufficientFee();
    providerInfo.accruedFeesInWei += providerInfo.feeInWei;
    _state.accruedPythFeesInWei += (SafeCast.toUint128(msg.value) -
        providerInfo.feeInWei);
...
}
```

Figure 12.1: Fees are credited immediately upon submission of a request. (pyth-crosschain/target_chains/ethereum/contracts/contracts/entropy/Entropy.sol#176–198)

Because fees are credited to the provider immediately, there is little financial incentive to keep the Fortuna service online.

Exploit Scenario

Alice registers a provider with the Entropy service, setting a 0.01 ETH fee for entropy. As a cost saving measure, Alice does not pay for redundant or high-availability hosting infrastructure for the Fortuna service.

An NFT protocol uses Alice as the entropy provider for a randomized mint. Alice's Fortuna server goes down for several hours, causing users to miss the 256-block revelation window for block hash–based entropy. Alice, however, still collects the fees from the randomness that she never provided.

Recommendations

Short term, lock user funds upon submission of a `request`, but credit the funds to the provider only upon completion of the `reveal` operation.

Long term, consider allowing providers to submit entropy seeds on their own behalf and collect fees if the user goes offline.

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**
Not Applicable	We did not identify a specific exploit scenario for this finding.
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. Automated Testing

This section describes the setup of the automated analysis tools used during this audit.

Slither

We used Slither to detect common issues and anti-patterns in the codebase. Slither discovered multiple low- and high-severity issues, such as TOB-ENTR-1, TOB-ENTR-2, and TOB-ENTR-4. Integrating Slither into the project's testing environment could help find similar issues and improve the overall quality of the smart contracts' code.

```
pip3 install slither-analyzer
slither .
```

Figure C.1: The commands used to install and run `slither-analyzer`

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

Clippy

The Rust linter Clippy can be installed using `rustup` by running the command `rustup component add clippy`. Invoking `cargo clippy` in the root directory of the project runs the tool.

cargo-llvm-cov

The `cargo-llvm-cov` Cargo plugin is used to generate LLVM source-based code coverage data. The plugin can be installed via the command `cargo install cargo-llvm-cov`. To run the plugin, run the command `cargo llvm-cov` in the crate root directory.

Necessist

The Necessist framework iteratively removes statements and method calls from tests and then runs them. If a test passes with a statement or method call removed, it could indicate a problem in the test or in the code being tested. Necessist found a total of 76 tests that pass with a statement or method call removed, four of which were in tests for in-scope contracts.

```
cargo install necessist
necessist --framework foundry
```

Figure C.2: The commands used to install and run Necessist

Semgrep

The semgrep tool identifies known anti-patterns and potential security vulnerabilities based on a library of lints. Semgrep can be installed with `brew install semgrep` or similar. We ran the tool with the standard Rust lints using `semgrep --config "p/rust"`.

D. Code Quality Recommendations

This appendix contains findings that are not directly related to security issues but could improve clarity, robustness, or auditability.

- The Fortuna Rust package currently reports 22 warnings when using `cargo clippy` with default settings. We recommend resolving all the warnings, most of which can be resolved using `cargo clippy --fix`, and incorporating a Clippy pass in the CI. This will ensure idiomatic Rust use throughout the codebase and prevent accumulation of potentially error-prone code.
- The `Entropy` and `Executor` contracts use a floating pragma definition. We recommend picking a specific version of the compiler to guarantee the contracts will be deployed with the intended compiler version.
- The `foundry.toml` configuration file specifies that the contracts will be compiled using the latest Solidity compiler version, 0.8.23. Using the latest compiler version is not recommended since it could introduce new and currently undiscovered bugs.
- Passing a `chainLength` parameter of zero to the `register` function of the `Entropy` contract will cause the function to revert with the custom error `AssertionFailure`, which could be made more descriptive.
- The `ProviderInfo` struct of the `EntropyStructs` contract does not fully use storage slot packing, so it uses more storage slots than necessary. The struct could be optimized by reordering the variable definitions.
- The `getPythFee` function of the `Entropy` contract is not defined in the `IEntropy` interface.
- The `withdraw` function of the `Entropy` contract and the `execute` function of the `Executor` contract do not emit events, which increases the difficulty of monitoring these actions.
- The `owner` input variable shadows an existing variable in the `EntropyUpgradeable` contract. Update the input variable to avoid variable name shadowing.
- The `register` function of the `Entropy` contract validates that the `chainLength` input variable is not equal to zero, but requesting entropy from a provider that registers with a `chainLength` of one will cause the request to revert. Consider updating the condition so that registering with a `chainLength` of less than two will revert.

- The protocol uses outdated OpenZeppelin dependencies that contain known vulnerabilities. Consider updating the OpenZeppelin dependencies to v4.9.5.

E. Incident Response Recommendations

This section provides recommendations on formulating an incident response plan.

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

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