IRONNODE

ironnode.io

Feed Protocol

Solana Program Security Audit

by **IronNode** from **April 30 - May 15, 2024**

Executive overview

1 Executive Overview

1.1 Introduction

Feed protocol provides fast, cost-effective access to secure and efficient randomness for blockchain applications.

In response to a request from Feed Protocol, IronNode, conducted a thorough security audit of the Feed Protocol to ensure its integrity, security, and user trustworthiness. The audit was carried out from April 30th, 2024 to May 15th, 2024. After the findings shared with the customer, IronNode carried out a verifying security audit for the Feed Protocol.

1.2 Audit Summary

The security audit team at IronNode was allocated two weeks to conduct an extensive review of the Feed Protocol. A dedicated team, including a lead security engineer with profound expertise in blockchain technology, smart contract security, and cybersecurity, conducted the audit.

1.2.1 Objectives of the Audit

- Ensure the Robustness of Business Logic: Validate the underlying business logic of Feed Protocol for any security flaws that could be exploited maliciously.
- Fee Structure Integrity: Verify that the one-time fee model is implemented securely and operates as intended without hidden risks.
- Smart Contract Examination: Conduct in-depth testing of the smart contracts to identify potential vulnerabilities, such as reentrancy attacks, overflow bugs, and improper exception handling.
- User Interaction Security: Evaluate the security protocols concerning user interactions with the platform to ensure that sensitive user data is handled securely.
- External Dependency Security: Review all external libraries and dependencies used in the protocol to confirm they do not introduce security vulnerabilities.
- Audit Trails and Monitoring: Check the adequacy of the logging and monitoring mechanisms that help in identifying and mitigating potential threats in real-time.

1.3 Scope

1.3.1 Code Repositories

feed-protocol
Branch: main
Commit ID: https://github.com/MintLabsDev/feed-protocol/commit/656cdefe36e4106a5ae3f6073b23cc8d8938f5f0

1.4 Assessment summary & findings overview

CRITICAL	HIGH	MEDIUM	LOW	INFORMATIONAL
0	0	3	3	0

Security analysis	Risk Level	Remediation date
Non-Present Fallback Addresses of Off-chain Components	MEDIUM	RESOLVED
Immutable Admin Address	MEDIUM	RESOLVED
Usage of Unchecked Arithmetics	MEDIUM	RESOLVED
Missing Account Flag Checks	LOW	RESOLVED
Unrecoverable Errors	LOW	RESOLVED
Inconsistency in Rent Calculations	LOW	RESOLVED

Application Flow Analysis

2 Feed Protocol

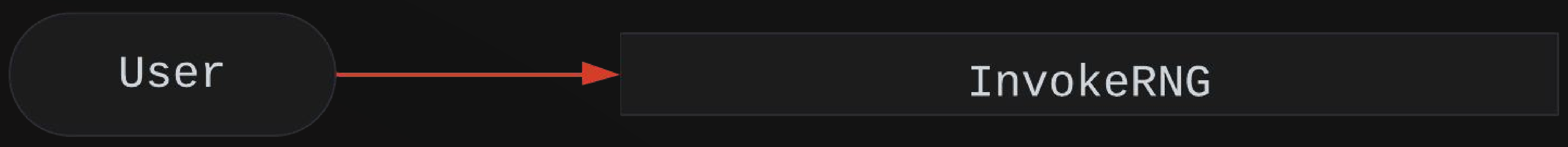
This section of the report provides an in-depth analysis of the application flow for the Feed Protocol contract. The analysis covers the logical flow of the code, with an emphasis on two primary roles: User and Admin.

2.1 User Role

The user role encompasses wallet addresses that interact with the Feed Protocol.

2.1.1 Functions

- InvokeRNG: Allows users to generate a pseudo random number using pyth price feeds, clock and slot number.

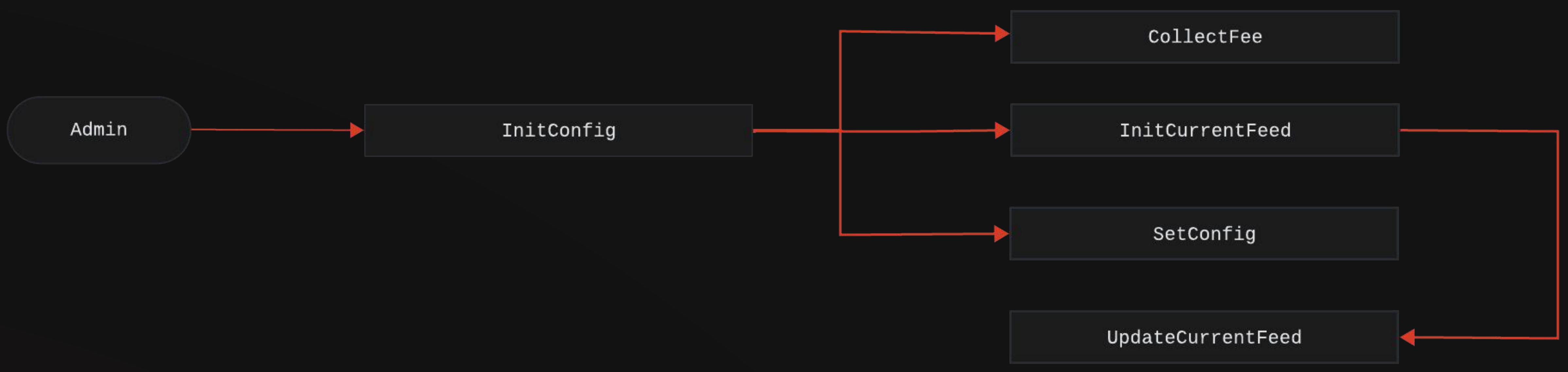


2.2 Admin Role

The admin role is the entity responsible for managing configuration settings and collecting fee over the Feed Protocol.

2.2.1 Functions

- InitConfig: Allows admin to initialize the passed config_account account. Can only be called once.
- SetConfig: Allows admin to update the passed config_account account.
- CollectFee: Allows admin to collect fee from the passed fee_account account.
- InitCurrentFeed: Allows admin to initialize the passed current_feed_account account. Can only be called once.
- UpdateCurrentFeed: Allows admin to update the passed current_feed_account account.



Findings

3 Findings

3.1 Non-Present Fallback Addresses Of Off-Chain Components (Resolved)

Relying only on Pyth Oracles for input data, creates single point of failure. Any disruptions in Pythnet or Wormhole, such as the incident in January 2024, could delay or halt the number generation processes.

3.1.1 Risk Level

Medium

3.1.2 Recommendation

Implement fallback mechanisms to alternate oracles to ensure continuous operation during disruptions.

3.2 Immutable Admin Address (Resolved)

The admin address stored in the authority_address variable is hard-coded at multiple points and should be stored in a manner that allows for reinitialization. When an admin change is necessary, the inability to effectively transfer contract management leads to operational risks.

3.2.1 Risk Level

Medium

3.2.2 Recommendation

Implement fallback mechanisms to alternate oracles to ensure continuous operation during disruptions.

- Allow for reinitialization of the admin address to improve maintainability.
- Store fee_account and authority_address in a configurable account to change it when it's necessary.

3.3 Usage Of Unchecked Arithmetics (Resolved)

The lack of overflow and underflow checks in arithmetic operations can lead to unexpected behavior, including but not limited to incorrect calculations, contract logic failures, or vulnerabilities that could potentially be exploited by attackers to manipulate the contract's state or logic in unintended ways. In the context of smart contracts, such vulnerabilities are particularly concerning as they could lead to financial loss and compromise contract integrity.

3.3.1 Risk Level

Medium

3.3.2 Recommendation

Utilize `checked_*` functions (or their alternatives) to ensure operations are safe and return error messages in case of failure. Handling errors improve the contract's reliability.

Example:

```
offset1 =
vec_for_offset_randomization[(offset_randomization_number_array[0].checked_sub(49)).ok_or(F
eedError::ArithmeticError)? as usize];
```

Unit testing helps discovering this kind of issues before compiling a release version.

3.4 Missing Account Flag Checks (Resolved)

Several functions lack necessary account flag checks. Furthermore, the program does not verify Pyth account addresses against those specified on the Pyth website, which can lead to using incorrect or unauthorized data sources.

3.4.1 Risk Level

Low

3.4.2 Code location

- In init_current_feed, the current_feed address does not check for writable and owner flags.
- In init_collect_fee, the fee_account address does not check for writable and owner flags.
- In collect_fee, the authority address lacks writable checks, and fee_account lacks signer and owner flag checks.
- In randomize, the payer account lacks signer and writable checks, the temporary account lacks signer checks, and fee_collect account lacks writable checks.

3.4.3 Recommendation

- Ensure all necessary account flag checks are implemented in the respective functions.
- Verify Pyth account addresses against the official sources specified on the Pyth website to ensure the integrity and authenticity of the data being used.
- Consider using the Anchor framework to manage these checks and control vulnerabilities like Account Cosplay.

3.5 Unrecoverable Errors (Resolved)

The usage of panic!() macros at multiple points in the code generates unrecoverable errors. Unrecoverable errors result in ambiguity and a poor user experience.

3.5.1 Risk Level

Low

3.5.2 Recommendation

Replace panic!() macros with readable error messages to ensure better error handling and system stability.

3.6 Inconsistency In Rent Calculations (Resolved)

Pre-calculating rent values can lead to incorrect or incomplete results. This approach might not accurately reflect the required balance for different account sizes, potentially causing account creation or transaction failures. Accurate rent calculation is crucial for maintaining the stability and reliability of the program.

3.6.1 Risk Level

Low

3.6.2 Code location

```
src/processor.rs (Line 502)
```

```
    pub fn collect_fee(accounts: &[AccountInfo]) -> ProgramResult {
        ...
        ...
        ...
        let value = **fee_account.lamports.borrow_mut() - feed.rent;
        ...
        ...
    }
```

```
src/processor.rs (Line 79-85)
```

```
pub fn randomize(accounts: &[AccountInfo], program_id: &Pubkey) -> ProgramResult {
    ...
    ...
        let create_ix = system_instruction::create_account(
            payer.key,
            temp.key,
            feed.fee + feed.rent,
            0,
            program_id,
        );
    ...
    ...
    }
```

3.6.3 Recommendation

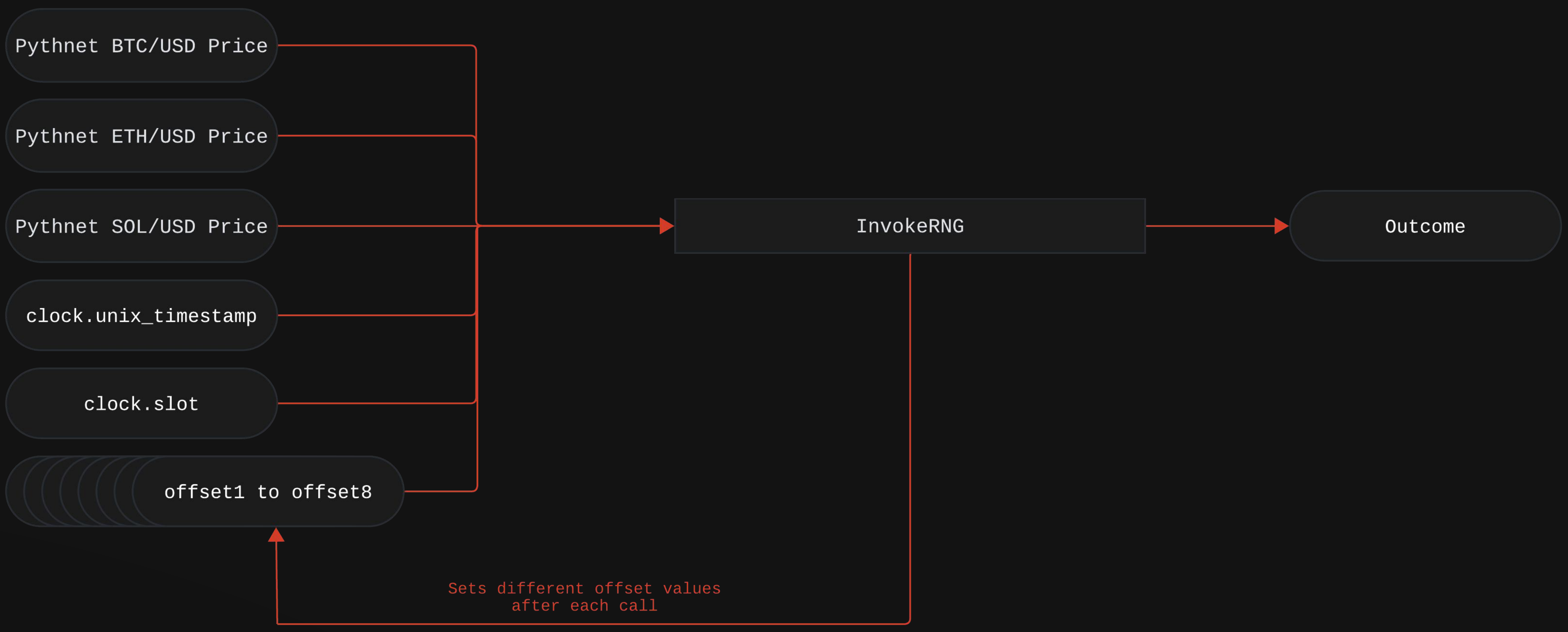
Use the minimum_balance function from solana_program::rent::Rent to calculate rent accurately based on the account size.

Attack Vector Analysis

4 Attack Vector Analysis

4.1 Predictable Randomness

An attacker attempts to predict the RNG outcome by exploiting the deterministic nature of the price data, timestamps, and slot numbers.



4.1.1 Risk Analysis

The price data, timestamps, slots and offsets are publicly accessible, the attacker can leverage this information to simulate and predict RNG outcomes.

New blocks can be proposed approximately every 400 milliseconds in Solana, and the Feed Protocol utilizes an offset mechanism to improve entropy of the pRNG system. This combined with account locks over the CurrentFeed accounts, makes predicting the outcome of this transaction extremely challenging in practice, since:

1. The offsets used in the RNG logic are updated with each transaction, adding a layer of unpredictability.
2. The exact timing of when a transaction is included in a block is difficult to control precisely, even if the attacker submits transactions rapidly.
3. The account locks over the CurrentFeed accounts prevent multiple transactions from being processed simultaneously, limiting the attacker's ability to manipulate or predict the RNG outcome.

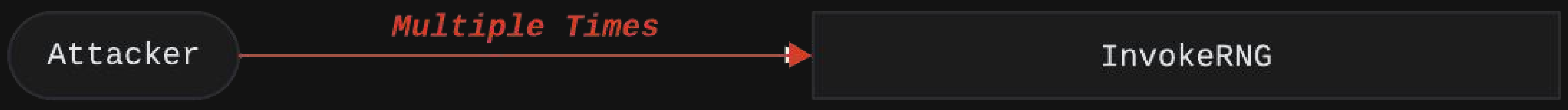
While theoretically possible, the combination of these factors makes it highly improbable for an attacker to consistently predict the RNG outcome in real-world scenarios.

However, the risk of a predictable outcome increases when the protocol is stale, as the offset mechanism becomes less effective in such cases. Constant activity of the protocol helps maintain the security of the outcome.

To further enhance the entropy of the RNG operation, the output of the Feed Protocol could be combined with different slicing mechanisms or modular arithmetic by the client. This additional step can add an extra layer of entorpy to the final result.

4.2 Re-entrancy and Race Condition

An attacker attempts to exploit the timing and state updates of the RNG protocol by sending multiple transactions in rapid succession, causing race conditions or re-entrancy issues.



1. Attacker waits for a period of high network congestion or slow transaction processing.
2. Attacker submits multiple transactions that call the `randomize` instruction almost simultaneously.
3. Attacker aims to exploit any delay or inconsistency in the account state updates to gain an advantage in generating predictable RNG outcomes or manipulating the state.

4.2.1 Risk Analysis

Attacker exploits this to execute multiple instructions before the account state is fully updated, leading to inconsistent or manipulated outcomes. High transaction volume might cause delays or race conditions in updating the account states, leading to potential exploits.

Since there'll be account locks over the CurrentFeed account, and due to Solana's depth restriction of CPIs, this vector is safe. Atomic and consistent state updates will prevent race conditions and re-entrancy issues.

IRONNODE

ironnode.io

Thank you for choosing us!