// Security Assessment

06.26.2026 - 06.29.2026

Redstone Oracle Integration Audit

Blueprint Finance

HALBORN

Redstone Oracle Integration Audit - Blueprint Finance

Prepared by: HALBORN

Last Updated 07/13/2026

Date of Engagement: June 26th, 2026 - June 29th, 2026

Summary

100% OF ALL REPORTED FINDINGS HAVE BEEN ADDRESSED

ALL FINDINGS	CRITICAL	HIGH	MEDIUM	LOW	INFORMATIONAL
5	0	0	1	1	3

TABLE OF CONTENTS

1. Introduction

The `Glow team` engaged `Halborn` to conduct a security assessment of set of changes of their **Margin, Margin-Pool and Vault Solana programs** beginning on June 26th, 2026, and ending on June 29th, 2026. The security assessment was scoped to the Solana Programs provided in **glow-v1** GitHub repository. Commit hashes and further details can be found in the Scope section of this report.

The implemented changes include:

- Added support for the Redstone price oracle.
- Updated Metaplex token metadata creation to support both the legacy and Token2022 token programs.
- Restricted operator admin assignment to the vault authority.
- Enforced that only the airspace authority can create a vault.
- Fixed a vulnerability where a liquidator could borrow on behalf of a liquidated user.
- Fixed a vulnerability where stale collateral could invalidate collateral checks.

2. Assessment Summary

`Halborn` was provided 2 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 Programs.
- Ensure that smart contract functionality operates as intended.

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

- `Validate the RedStone's feed_id during price feed creation.`
- `Implement price manipulation protection for Chainlink and RedStone oracle price feeds.`
- `Calculate the combined oracle staleness window as minimum of both oracle values.`
- `Monitor for Redstone oracle heartbeat health and define a protocol-level response for prolonged oracle outages.`
- `Remove dead code.`

3. Test Approach And Methodology

Halborn performed a combination of a manual review of the source code and automated security testing to balance efficiency, timeliness, practicality, and accuracy in regard to the scope of the program assessment. While manual testing is recommended to uncover flaws in business logic, processes, and implementation; automated testing techniques help enhance coverage of programs and can quickly identify items that do not follow security best practices.

The following phases and associated tools were used throughout the term of the assessment:

- Research into the architecture, purpose, and use of the platform.
- Manual program source code review to identify business logic issues.
- Mapping out possible attack vectors
- Thorough assessment of safety and usage of critical Rust variables and functions in scope that could lead to arithmetic vulnerabilities.
- Scanning dependencies for known vulnerabilities (`cargo audit`).
- Local runtime testing (`solana-test-framework`)

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
Attack Cost (AC)	Low (AC:L) Medium (AC:M) High (AC:H)	1 0.67 0.33

EXPLOITABILITY METRIC (M_E)	METRIC VALUE	NUMERICAL VALUE
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

IMPACT METRIC (M_I)	METRIC VALUE	NUMERICAL VALUE
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
Informational	0 - 1.9

5. SCOPE

REPOSITORY

(a) Repository: glow-v1

(b) Assessed Commit ID: 3c40024

(c) Items in scope:

- programs/margin-pool/src/instructions/margin_borrow_v2.rs
- programs/margin-pool/src/lib.rs
- programs/margin/src/adapter.rs
- programs/margin/src/instructions/liquidate_end.rs
- programs/margin/src/instructions/liquidator_invoke.rs
- programs/margin/src/instructions/oracle/create_price_feed.rs
- programs/margin/src/instructions/oracle/refresh_price_feed.rs
- programs/margin/src/lib.rs
- programs/margin/src/state/account.rs
- programs/margin/src/state/price_feed.rs
- programs/vault/src/instructions/admin/assign_vault_operator_admin.rs
- programs/vault/src/instructions/admin/create_vault.rs
- programs/vault/src/instructions/admin/update_share_token_metadata.rs

FILE

(a) Submitted File: glow-v1-audits-2026-06-redstone.zip

(b) Items in scope:

- /glow-v1-audits-2026-06-redstone/programs/margin/src/adapter.rs
- /glow-v1-audits-2026-06-redstone/programs/margin/src/lib.rs
- /glow-v1-audits-2026-06-redstone/programs/margin/Cargo.toml
- /glow-v1-audits-2026-06-redstone/programs/margin/Xargo.toml
- /glow-v1-audits-2026-06-redstone/Anchor.toml
- /glow-v1-audits-2026-06-redstone/Cargo.toml
- /glow-v1-audits-2026-06-redstone/docs/margin-rust/ .gitkeep
- /glow-v1-audits-2026-06-redstone/docs/index.html
- /glow-v1-audits-2026-06-redstone/docs/solayer-staking-flow.md
- /glow-v1-audits-2026-06-redstone/docs/style.css
- /glow-v1-audits-2026-06-redstone/docs/test-coverage.md
- /glow-v1-audits-2026-06-redstone/docs/token-configs-ci.md
- /glow-v1-audits-2026-06-redstone/programs/margin/src/instructions/oracle/create_price_feed.rs

- /glow-v1-audits-2026-06-redstone/programs/margin/src/instructions/oracle/refresh_price_feed.rs
- /glow-v1-audits-2026-06-redstone/programs/margin/src/instructions/liquidate_end.rs
- /glow-v1-audits-2026-06-redstone/programs/margin/src/instructions/liquidator_invoke.rs
- /glow-v1-audits-2026-06-redstone/programs/margin/src/state/price_feed.rs
- /glow-v1-audits-2026-06-redstone/programs/margin/src/state/account.rs
- /glow-v1-audits-2026-06-redstone/programs/margin-pool/src/instructions/margin_borrow_v2.rs
- /glow-v1-audits-2026-06-redstone/programs/margin-pool/src/lib.rs
- /glow-v1-audits-2026-06-redstone/programs/vault/src/instructions/admin/assign_vault_operator_admin.rs
- /glow-v1-audits-2026-06-redstone/programs/vault/src/instructions/admin/create_vault.rs
- /glow-v1-audits-2026-06-redstone/programs/vault/src/instructions/admin/update_share_token_metadata.rs

REMEDIATION COMMIT ID:

- 8cfc0c7
- d24b9f1
- eb7edbe
- c88b2f9
- 9502453

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

6. ASSESSMENT SUMMARY & FINDINGS OVERVIEW

CRITICAL	HIGH	MEDIUM	LOW
0	0	1	1

INFORMATIONAL
3

SECURITY ANALYSIS	RISK LEVEL	REMEDIATION DATE
CONFIDENCE INTERVAL CHECK PERMANENTLY BYPASSED FOR REDSTONE AND CHAINLINK ORACLES	MEDIUM	SOLVED - 07/02/2026
COMBINED ORACLE FEED STALENESS WINDOW INCORRECTLY USES MAXIMUM INSTEAD OF MINIMUM	LOW	SOLVED - 07/02/2026
STALE REDSTONE-BACKED COLLATERAL PRICES BLOCK LIQUIDATION VIA VERIFY_NO_STALE_COLLATERAL	INFORMATIONAL	SOLVED - 07/06/2026
INSUFFICIENT REDSTONE PRICE FEED VALIDATION MAY CAUSE DENIAL OF SERVICE	INFORMATIONAL	SOLVED - 07/02/2026
DEAD CODE IN LIQUIDATOR_INVOKE	INFORMATIONAL	SOLVED - 07/02/2026

7. FINDINGS & TECH DETAILS

7.1 CONFIDENCE INTERVAL CHECK PERMANENTLY BYPASSED FOR REDSTONE AND CHAINLINK ORACLES

// MEDIUM

Description

For Redstone and Chainlink oracle feeds, `PriceChangeInfo` is constructed with `confidence = 0` and `ema = value` (spot price). This hardcoded combination permanently disables the `MAX_ORACLE_CONFIDENCE` check in `to_price_info()`, which is designed to reject prices when the confidence interval is too wide relative to the EMA, indicating high price uncertainty or potential manipulation.

Since the check evaluates `(c / ema) > max_confidence` — where `max_confidence = 5%` — substituting the hardcoded values yields `(0 / ema) > 5%`, which always evaluates to `false`. As a result, no Redstone or Chainlink price is ever rejected on confidence grounds. Furthermore, setting `ema = value` (spot price) undermines the EMA-based sanity check as well — a manipulated spot price simply sets its own EMA, making the check circular and effectively meaningless.

As a result, a manipulated or incorrect oracle price could allow an attacker to borrow funds against overvalued collateral, withdraw more assets than entitled, or avoid liquidation. Conversely, an artificially depressed price could enable a malicious liquidator to unfairly liquidate healthy positions.

programs/margin/src/adapter.rs

Copy Code

```rust
pub fn to_price_info(self, unix_timestamp: UnixTimestamp) -> PriceInfo {
    let max_confidence = Number128::from_bps(MAX_ORACLE_CONFIDENCE);

    let ema = Number128::from_decimal(self.ema, self.exponent);
    let confidence = Number128::from_decimal(self.confidence, self.exponent);

    if ema == Number128::ZERO {
        msg!("avg price cannot be zero");
        return PriceInfo::new_invalid();
    }

    match (confidence, self.publish_time) {
        (c, _) if (c / ema) > max_confidence => {
            msg!("price confidence exceeding max");
            PriceInfo::new_invalid()
        }
        (_, publish_time) if (unix_timestamp - publish_time) > self.max_oracle_staleness() =>
            msg!(
                "price timestamp is too old/stale. published: {}, now: {} [{}], max staleness:
                publish_time,
                unix_timestamp,
                unix_timestamp - publish_time,
                self.max_oracle_staleness()
            );
            PriceInfo::new_invalid()
        }
        _ => PriceInfo::new_valid(self.exponent, self.value, self.publish_time as u64),
    }
}
```

Copy Code

```
fn redstone_price_change_info(
    expected_feed_id: &[u8; 32],
    account_feed_id: &[u8; 32],
    value: i64,
    decimals: u8,
    timestamp_ms: i64,
) -> Result<Self> {
    require!(
        account_feed_id == expected_feed_id,
        crate::ErrorCode::InvalidOracle
    );
    require!(
        decimals == crate::REDSTONE_EXPECTED_DECIMALS,
        crate::ErrorCode::OracleExponentMismatch
    );
    Ok(Self {
        value,
        // Redstone feeds do not provide a separate confidence
        confidence: 0,
        ema: value,
        // Redstone timestamps are in milliseconds; convert to seconds.
        publish_time: timestamp_ms / 1000,
        exponent: -(decimals as i32),
        max_oracle_staleness: MAX_REDSTONE_ORACLE_STALENESS as i64,
    })
}
```

Proof of Concept

Copy Code

```
#[tokio::test(flavor = "multi_thread")]
#[cfg_attr(not(feature = "localnet"), serial_test::serial)]
async fn extreme_price_comparison() -> Result<()> {
    let ctx = MarginTestContext::new("extreme_comparison", &[]).await?;

    // Get current clock
    let clock_data = ctx.rpc().get_account(&Clock::id()).await?.unwrap();
    let clock = bincode::deserialize::<Clock>(&clock_data.data)?;
    let current_time = clock.unix_timestamp;

    // Scenario: An extreme price event happens (e.g., flash crash, oracle manipulation)
    let normal_price = 100_000_000i64; // $100
    let extreme_price = 150_000_000i64; // $150 (50% spike)

    // For Pyth: the confidence interval would widen during volatility
    let pyth_high_conf = 10_000_000i64; // 10% confidence (would be rejected)
    let feed_id = [4u8; 32];

    let pyth_extreme = PriceUpdateV2 {
        write_authority: Pubkey::default(),
        verification_level: pyth_solana_receiver_sdk::price_update::VerificationLevel::Full,
        posted_slot: clock.slot,
        price_message: PriceFeedMessage {
            feed_id,
            price: extreme_price,
            conf: pyth_high_conf as u64,
            exponent: -8,
            publish_time: current_time,
            prev_publish_time: current_time - 10,
            ema_price: normal_price, // EMA hasn't caught up
            ema_conf: pyth_high_conf as u64,
        },
    };

    let pyth_price_change =
        PriceChangeInfo::try_from_pyth_pull(&pyth_extreme, &feed_id, &clock).unwrap();
    let pyth_price_info = pyth_price_change.to_price_info(current_time);
```

```
    // Pyth should reject due to high confidence
    assert!(
        !pyth_price_info.is_valid(),
        "Pyth should reject extreme price with high confidence"
    );

    // For Redstone: same extreme price, but confidence is always 0
    let redstone_feed_id = [5u8; 32];
    let redstone_price_change = PriceChangeInfo::redstone_price_change_info(
        &redstone_feed_id,
        &redstone_feed_id,
        extreme_price,
        8,
        current_time * 1000,
    )
    .unwrap();

    let redstone_price_info = redstone_price_change.to_price_info(current_time);

    // Redstone accepts the same extreme price (VULNERABILITY)
    assert!(
        redstone_price_info.is_valid(),
        "Redstone accepts extreme price due to bypassed confidence check"
    );

    println!("\n=== ASYMMETRIC PROTECTION CONFIRMED ===");
    println!("Scenario: Extreme price spike from $100 to $150 (50% increase)");
    println!("\nPyth Oracle:");
    println!("  - Price: ${:.2}", extreme_price as f64 / 1e8);
    println!(
        "  - Confidence: {}% (high volatility detected)",
        (pyth_high_conf as f64 / extreme_price as f64) * 100.0
    );
    println!("  - Result: REJECTED by MAX_ORACLE_CONFIDENCE check");
    println!("\nRedstone Oracle:");
    println!("  - Price: ${:.2}", extreme_price as f64 / 1e8);
    println!("  - Confidence: 0 (hardcoded, no volatility detection)");
    println!("  - Result: ACCEPTED (circuit breaker bypassed)");
    println!("\nThis asymmetry means Redstone/Chainlink lack protection during:");
    println!("  - Flash crashes / price manipulation");
    println!("  - Extreme market volatility");
    println!("  - Oracle network anomalies");
    println!("========================================\n");

    Ok(())
}
```

```
--- STDOUT:              hosted-tests::oracle confidence_bypass::extreme_price_comparison ---

running 1 test
price confidence exceeding max

=== ASYMMETRIC PROTECTION CONFIRMED ===
Scenario: Extreme price spike from $100 to $150 (50% increase)

Pyth Oracle:
  - Price: $1.50
  - Confidence: 6.666666666666667% (high volatility detected)
  - Result: REJECTED by MAX_ORACLE_CONFIDENCE check

Redstone Oracle:
  - Price: $1.50
  - Confidence: 0 (hardcoded, no volatility detection)
  - Result: ACCEPTED (circuit breaker bypassed)

This asymmetry means Redstone/Chainlink lack protection during:
  - Flash crashes / price manipulation
  - Extreme market volatility
  - Oracle network anomalies
========================================

test confidence_bypass::extreme_price_comparison ... ok

test result: ok. 1 passed; 0 failed; 0 ignored; 0 measured; 7 filtered out; finished in 0.38s


--- STDERR:              hosted-tests::oracle confidence_bypass::extreme_price_comparison ---
```

BVSS

AO:A/AC:L/AX:H/R:N/S:U/C:N/A:N/I:C/D:C/Y:C (5.0)

Recommendation

Since the Redstone and Chainlink oracles feeds do not provide native confidence intervals and EMA value, it is recommended to implement alternative price manipulation detection, such as: maximum price deviation checks between consecutive updates; or circuit breakers based on time-weighted average prices.

Remediation Comment

SOLVED: The `Glow` team resolved this finding by implementing a maximum price deviation check for confidence-less oracles and rejecting non-positive price updates.

Remediation Hash

https://github.com/Blueprint-Finance/glow-v1/commit/8cfc0c72d42e2bd23be190a6dcce84ad1f07b7c6

7.2 COMBINED ORACLE FEED STALENESS WINDOW INCORRECTLY USES MAXIMUM INSTEAD OF MINIMUM

// LOW

Description

In `PriceChangeInfo::multiply()`, when two oracle feed prices — such as a Pyth quote feed and a Redstone redemption feed — are combined into a single price for a `GlowPriceFeed`, the `max_oracle_staleness` of the resulting `PriceChangeInfo` is set to `self.max_oracle_staleness.max(other.max_oracle_staleness)`, inheriting the most permissive staleness window of either component feed. For example, if the `quote_feed` is Pyth (`MAX_ORACLE_STALENESS = 60s`) and the `redemption_feed` is Redstone (`MAX_REDSTONE_ORACLE_STALENESS = 180s`), the combined price carries `max_oracle_staleness = 180s`. This allows the downstream `to_price_info()` staleness check to accept a combined price up to 180 seconds old — even though the Pyth component's own staleness guarantee is only 60 seconds.

This is semantically inconsistent: while `publish_time` is correctly set using `min()` — taking the older timestamp as the conservative choice — `max_oracle_staleness` uses `max()`, which is permissive. A correct implementation should use `min()` for the staleness window as well, ensuring the combined price is only considered valid if both components are within their respective freshness windows.

As a result, the protocol may accept a combined price based on a stale Pyth component that would otherwise be rejected, since the more permissive Redstone staleness window masks the freshness violation. This could allow borrowing against, or avoiding liquidation based on, an outdated price that no longer reflects current market conditions.

Using `max()` for `max_oracle_staleness` breaks the Pyth oracle's 60-second freshness guarantee, which is a security property rather than a suggestion. This creates a semantic contradiction with the conservative `min()` used for `publish_time` — the combined price may be flagged as 59 seconds old while still being accepted for up to 180 seconds. In volatile markets, this allows stale Pyth data to be used for borrowing, health checks, and liquidations, all of which may produce incorrect results. The `max()` approach provides no benefit — it only weakens the security guarantee. If a 180-second validity window is acceptable, only Redstone should be used; if Pyth's precision is required, its 60-second limit must be respected.

programs/margin/src/adapter.rs

Copy Code

```
472 pub fn multiply(&self, other: &Self) -> Result<Self> {
473     let price_value = Number128::from_decimal(self.value, self.exponent);
474     let price_ema = Number128::from_decimal(self.ema, self.exponent);
475     let price_conf = Number128::from_decimal(self.confidence, self.exponent);
476
477     let quote_value = Number128::from_decimal(other.value, other.exponent);
478     let quote_ema = Number128::from_decimal(other.ema, other.exponent);
479     let quote_conf = Number128::from_decimal(other.confidence, other.exponent);
480
481     // NOTE: this logic is duplicated from try_from_pyth_pull_redemption above.
482     let value_product = price_value
483         .safe_mul(quote_value)
484         .map_err(|_| error!(crate::ErrorCode::MathOpFailed))?;
```

```
        let value = value_product
            .try_as_i64(self.exponent)
            .ok_or(error!(crate::ErrorCode::MathOpFailed))?;
        // Confidence = (price_conf * quote_value) + (quote_conf * price_value)
        let confidence_product_a = price_conf
            .safe_mul(quote_value)
            .map_err(|_| error!(crate::ErrorCode::MathOpFailed))?;
        let confidence_product_b = quote_conf
            .safe_mul(price_value)
            .map_err(|_| error!(crate::ErrorCode::MathOpFailed))?;
        let confidence_sum = confidence_product_a
            .safe_add(confidence_product_b)
            .map_err(|_| error!(crate::ErrorCode::MathOpFailed))?;
        let confidence = confidence_sum
            .try_as_u64(self.exponent)
            .ok_or(error!(crate::ErrorCode::MathOpFailed))?;
        let ema_product = price_ema
            .safe_mul(quote_ema)
            .map_err(|_| error!(crate::ErrorCode::MathOpFailed))?;
        let ema = ema_product
            .try_as_i64(self.exponent)
            .ok_or(error!(crate::ErrorCode::MathOpFailed))?;
        let publish_time = self.publish_time.min(other.publish_time);
        let max_oracle_staleness = self.max_oracle_staleness.max(other.max_oracle_staleness);

        Ok(Self {
            value,
            confidence,
            ema,
            publish_time,
            exponent: self.exponent,
            max_oracle_staleness,
        })
    }
```

BVSS

AO:A/AC:L/AX:H/R:N/S:U/C:N/A:N/I:M/D:C/Y:N (3.7)

Recommendation

It is recommended to change the `max_oracle_staleness` calculation in `multiply()` to use `min()` instead of `max()`, so the combined price inherits the strictest staleness requirement of its components. If the resulting staleness window is too restrictive for practical use, increasing the Pyth staleness threshold could be considered to ensure both feeds can work correctly together.

Remediation Comment

SOLVED: The `Glow` team resolved this finding by using the `min()` so the combined price inherits the strictest staleness requirement of its components.

Remediation Hash

https://github.com/Blueprint-Finance/glow-v1/commit/d24b9f1c65d4813bb98315a9a3ee4049b4d56f3f

7.3 STALE REDSTONE-BACKED COLLATERAL PRICES BLOCK LIQUIDATION VIA VERIFY_NO_STALE_COLLATERAL

// INFORMATIONAL

Description

The `liquidator_invoke_handler` calls `verify_no_stale_collateral()` on both the start and end valuations. If any collateral position has a stale cached price — i.e., `price_quote_age = current_time - position.price.timestamp > MAX_PRICE_QUOTE_AGE = 30s` — the liquidation call reverts with `ErrorCode::StalePositions`.

For accounts where collateral is priced via a Redstone-backed `GlowPriceFeed`, the cached `position.price.timestamp` is derived from the Redstone oracle's data-package timestamp (`redstone.timestamp / 1000`). Since Redstone pushers update on a heartbeat schedule, there is a guaranteed liveness window during which the position price is considered stale. According to the Redstone Push Price Feeds explorer, the heartbeat for all Solana price accounts is 24 hours. While the observed refresh rate for most feeds may be in the order of seconds, this cannot be considered a guarantee, as price feeds are only guaranteed to update at the heartbeat frequency.

This creates a systemic liquidation denial of service: any account whose Redstone-priced collateral has not been refreshed within the last 30 seconds cannot be liquidated. In the worst case, this could allow unhealthy positions to avoid liquidation for extended periods, exposing the protocol to bad debt accumulation and potential insolvency.

programs/margin/src/instructions/liquidator_invoke.rs

Copy Code

```
165 let liquidation = &mut ctx.accounts.liquidation.load_mut()?.state;
166 let end_value = update_and_verify_liquidation(
167     &*ctx.accounts.margin_account.load()?,
168     liquidation,
169     start_value,
170 )?;
```

programs/margin/src/instructions/liquidator_invoke.rs

Copy Code

```
182 fn update_and_verify_liquidation(
183     margin_account: &MarginAccount,
184     liquidation: &mut Liquidation,
185     start_value: Valuation,
186 ) -> Result<Valuation> {
187     let end_value = margin_account.valuation(sys().unix_timestamp())?;
188     end_value.verify_no_stale_collateral()?;
```

BVSS

AO:A/AC:L/AX:H/R:P/S:U/C:N/A:C/I:N/D:N/Y:N (1.6)

Recommendation

To address this finding, it is recommended to implement off-chain monitoring for Redstone oracle heartbeat health and define a protocol-level response — such as switching to an alternative price feed — for prolonged oracle outages that affect liquidatability.

Remediation Comment

SOLVED: The `Glow` team resolved this finding by ensuring that `PriceInfo` carries the oracle's maximum staleness, which is typically greater than 30 seconds. The team also stated that they are adding an off-chain service to monitor Redstone feed staleness, and that based on feed performance after deployment, additional steps may be taken — such as considering alternative oracles.

Remediation Hash

https://github.com/Blueprint-Finance/glow-v1/commit/eb7edbe65de00a0a128f849351dc49927a2124c4

7.4 INSUFFICIENT REDSTONE PRICE FEED VALIDATION MAY CAUSE DENIAL OF SERVICE

// INFORMATIONAL

Description

The `create_price_feed` instruction allows an airspace authority to create a new price feed account. A recent update added the ability to store a Redstone Oracle price feed address along with a corresponding feed ID.

However, the instruction does not deserialize the Redstone price feed data and does not verify that the provided `feed_id` corresponds to the account's `feed_id` field, nor does it validate the feed's PDA, which is derived from the `feed_id` value as follows:

Copy Code

```
type FeedIdBs = [u8; 32];  // right-zero-padded byte-list representing the feed-id ascii bytes
const PROGRAM_ID: &str = "REDSTBDUecGjwXd6YGPzHSvEUBHQqVRfCcjUVgPiHsr";

fn make_price_seed() -> [u8; 32] {
    let mut seed = [0u8; 32];
    seed[0..5].copy_from_slice(b"price");

    seed
}

fn price_feed_account_pubkey(feed_id: &FeedIdBs) -> (Pubkey, u8) {
    Pubkey::find_program_address(&[
        &make_price_seed(), feed_id
    ], &Pubkey::from_str(PROGRAM_ID).unwrap())
}
```

Providing an incorrect `feed_id` will pass all validations and the instruction will return successfully. However, this will result in a denial of service of the `refresh_price_feed` instruction, as the `redstone_price_change_info` helper function validates that the stored `OracleFeed::feed_id` matches the actual Redstone price feed ID, which will fail. Furthermore, the program does not provide a direct way to correct a misconfigured price feed account — the only option without a program upgrade is to create a new price feed and reconfigure the token. While the stored `feed_id` is not directly used by the on-chain program, this inconsistency may confuse downstream off-chain consumers and lead to unexpected behavior.

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

Copy Code

```
114 fn validate_oracle_feed_configuration(feed: &crate::state::OracleFeed) -> Result<()> {
115     match feed {
116         crate::state::OracleFeed::Empty(_) => Ok(()),
117         crate::state::OracleFeed::PythPull { feed_id } => {
118             require!(*feed_id != [0u8; 32], crate::ErrorCode::InvalidOracleFeedId);
119             Ok(())
120         }
121         crate::state::OracleFeed::Chainlink { address } => {
122             require!(
123                 *address != Pubkey::default(),
124                 crate::ErrorCode::InvalidOracleFeedId
125             );
126
```

```
            Ok(())
        }
        crate::state::OracleFeed::Redstone { feed_id, address } => {
            require!(*feed_id != [0u8; 32], crate::ErrorCode::InvalidOracleFeedId);
            require!(
                *address != Pubkey::default(),
                crate::ErrorCode::InvalidOracleFeedId
            );
            Ok(())
        }
    }
}
```

programs/margin/src/adapter.rs

Copy Code

```
fn redstone_price_change_info(
    expected_feed_id: &[u8; 32],
    account_feed_id: &[u8; 32],
    value: i64,
    decimals: u8,
    timestamp_ms: i64,
) -> Result<Self> {
    require!(
        account_feed_id == expected_feed_id,
        crate::ErrorCode::InvalidOracle
    );
    require!(
        decimals == crate::REDSTONE_EXPECTED_DECIMALS,
        crate::ErrorCode::OracleExponentMismatch
    );
    Ok(Self {
        value,
        // Redstone feeds do not provide a separate confidence
        confidence: 0,
        ema: value,
        // Redstone timestamps are in milliseconds; convert to seconds.
        publish_time: timestamp_ms / 1000,
        exponent: -(decimals as i32),
        max_oracle_staleness: MAX_REDSTONE_ORACLE_STALENESS as i64,
    })
}
```

BVSS

AO:S/AC:L/AX:L/R:P/S:U/C:N/A:C/I:M/D:N/Y:N (1.1)

Recommendation

To address this finding, it is recommended to validate the provided `feed_id` either by deserializing the Redstone price feed account or by validating the account's PDA.

Remediation Comment

SOLVED: The `Glow` team resolved this finding by ensuring that the provided `feed_id` matches the feed ID stored in the reference Redstone price account.

Remediation Hash

https://github.com/Blueprint-Finance/glow-v1/commit/c88b2f97376c07b6d78b0956c2da28458acfc8c3

7.5 DEAD CODE IN LIQUIDATOR_INVOKE

// INFORMATIONAL

Description

The `liquidator_invoke` instruction now prevents liquidators from borrowing on behalf of liquidated accounts by ensuring that no adapter-reported change is caused by a borrow.

If a `TokenBalanceChangeCause::Borrow` is detected, the instruction fails and returns an error. However, the subsequent code still contains `TokenBalanceChangeCause::Borrow` matches and filters that can no longer be satisfied, making them dead code that should be removed.

programs/margin/src/instructions/liquidator_invoke.rs

Copy Code

```
71  if token_changes
72      .iter()
73      .any(|c| c.change_cause == TokenBalanceChangeCause::Borrow)
74  {
75      msg!("Liquidators may not borrow on behalf of liquidated accounts");
76      return err!(crate::ErrorCode::LiquidationLostValue);
77  }
78
79  // Ensure the fee mint matches any repay/borrow token changes.
80  let has_repay_or_borrow = token_changes.iter().any(|c| {
81      matches!(
82          c.change_cause,
83          TokenBalanceChangeCause::Borrow | TokenBalanceChangeCause::Repay
84      )
85  });
86  let fee_mint_in_repay_or_borrow = token_changes.iter().any(|c| {
87      c.mint == ctx.accounts.liquidator_fee_mint.key()
88          && matches!(
89              c.change_cause,
90              TokenBalanceChangeCause::Borrow | TokenBalanceChangeCause::Repay
91          )
92  });
93
94  require!(
95      !has_repay_or_borrow || fee_mint_in_repay_or_borrow,
96      ErrorCode::InvalidLiquidatorFeeMint
97  );
98
99  // Determine the token changes of the relevant token to take a fee in.
100 let fee_relevant_changes = token_changes
101     .iter()
102     .filter(|c| {
103         // Relevant changes are only the changes in the tokens of the fee mint,
104         // and the changes that a liquidator is expected to make while liquidating.
105         c.mint == ctx.accounts.liquidator_fee_mint.key()
106             && [
107                 TokenBalanceChangeCause::Borrow,
108                 TokenBalanceChangeCause::Repay,
109                 TokenBalanceChangeCause::ExternalDecrease,
110                 TokenBalanceChangeCause::ExternalIncrease,
111             ]
112             .contains(&c.change_cause)
113     })
114     .collect::<Vec<_>>();
115 // The fee for swaps is based on the lower of the increase in the token and the repaid amount
116 let increases: i128 = fee_relevant_changes
117     .iter()
118     .map(|c| {
119         match c.change_cause {
120             TokenBalanceChangeCause::ExternalIncrease => c.tokens as i128,
121             // Offset increases
```

```
                TokenBalanceChangeCause::ExternalDecrease => -(c.tokens as i128),
                _ => 0,
            }
        })
        .sum();

    let repayments: i128 = fee_relevant_changes
        .iter()
        .map(|c| match c.change_cause {
            TokenBalanceChangeCause::Borrow => -(c.tokens as i128),
            TokenBalanceChangeCause::Repay => c.tokens as i128,
            _ => 0,
        })
        .sum();
```

BVSS

AO:A/AC:L/AX:M/R:N/S:U/C:N/A:N/I:N/D:N/Y:N (0.0)

Recommendation

To address this finding, it is recommended to remove the dead code.

Remediation Comment

SOLVED: The `Glow` team resolved this finding by removing the unreachable Borrow handling branches while keeping the `TokenBalanceChangeCause::Borrow` that rejects liquidator borrows unchanged.

Remediation Hash

https://github.com/Blueprint-Finance/glow-v1/commit/95024539d9d6db3712215f149fca942014f20bb3

8. AUTOMATED TESTING

Static Analysis Report

Description

Halborn used automated security scanners to assist with 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 auditors 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.

Cargo Audit Results

`No vulnerabilities were detected.`

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.