Veridise

Auditing Report

Hardening Blockchain Security with Formal Methods

FOR

Decentralized Oracle Integration

Veridise Inc.

August 21, 2024

► **Prepared For:**

Navi
https://naviprotocol.io/

► **Prepared By:**

Jon Stephens
Ajinkya Rajput
Evgeniy Shishkin

► **Contact Us:**

contact@veridise.com

► **Version History:**

June 28, 2024	V3
June 26, 2024	V2
June 21, 2024	V1
June 13, 2024	Initial Draft

Contents

Contents iii

1 Executive Summary 1

2 Project Dashboard 3

3 Audit Goals and Scope 5
- 3.1 Audit Goals . . . 5
- 3.2 Audit Methodology & Scope . . . 5
- 3.3 Classification of Vulnerabilities . . . 5

4 Vulnerability Report 7
- 4.1 Detailed Description of Issues . . . 8
  - 4.1.1 V-NOR-VUL-001: Staleness of oracle price can exceed expected bounds . 8
  - 4.1.2 V-NOR-VUL-002: Cannot update all prices via API . . . 10
  - 4.1.3 V-NOR-VUL-003: Centralization Risk . . . 11
  - 4.1.4 V-NOR-VUL-004: Users can update prices in their favor . . . 12
  - 4.1.5 V-NOR-VUL-005: No validation when changing oracle id . . . 13
  - 4.1.6 V-NOR-VUL-006: No check that oracle is not both primary and secondary 14
  - 4.1.7 V-NOR-VUL-007: No validation when updating Oracle's update_interval 16
  - 4.1.8 V-NOR-VUL-008: No check if cast may overflow . . . 18
  - 4.1.9 V-NOR-VUL-009: Changes to update_interval can introduce unfairness 19
  - 4.1.10 V-NOR-VUL-010: Pyth price confidence not considered . . . 20
  - 4.1.11 V-NOR-VUL-011: Little validation performed on price feed creation . . . 21
  - 4.1.12 V-NOR-VUL-012: No decimal validation on price update . . . 23
  - 4.1.13 V-NOR-VUL-013: No validation of decimals on conversion . . . 24

1 Executive Summary

From June 7, 2024 to June 13, 2024, Navi engaged Veridise to review the security of their Decentralized Oracle Integration. The review covered an update to Navi's oracle module to make use of decentralized oracles. Veridise conducted the assessment over 3 person-weeks, with 3 security analysts reviewing code over 1 weeks on commit `cb7ea49`. The auditing strategy involved a tool-assisted analysis of the source code performed by Veridise security analysts as well as extensive manual auditing.

Project summary. The security assessment covered the new decentralized oracle module for the Navi protocol. Previous versions of Navi made use of a centralized oracle which relied on admins to frequently update token prices. The update in scope of this audit allows the Navi oracle to receive price updates from two decentralized oracle providers: Pyth and Supra. To do so, users or admins must call a new API which will fetch prices from the two decentralized oracles. As both of these oracle providers have relatively new infrastructure deployed on SUI, Navi checks the two oracle prices against each other along with other sanity checks. As long as these pass, the price is pushed to Navi's oracle module which may then be consumed by the remainder of the lending protocol. It should be noted that while the decentralized oracle is intended to be the main source of pricing data, the Navi admins retain their ability to update the oracle manually as well.

Code assessment. The Navi developers provided the source code of the Decentralized Oracle Integration contracts for review. The new oracle code is entirely original code written by the Navi developers. To facilitate the Veridise security analysts understanding of the code, the Navi developers provided a document describing the intended behavior of the oracle, and a test deployment used by the developers to ensure the end-to-end product was behaving as intended. Additionally, the code contained some in-line comments on structs and functions. The delivered source code contained a test-suite, but it appeared that some functionality was difficult to test due to the dependence on the oracle providers.

Summary of issues detected. The audit uncovered 13 issues, 1 of which is assessed to be of high or critical severity by the Veridise security analysts. Specifically, V-NOR-VUL-001 identifies the potential for listed prices to exceed the expected bound, potentially allowing very stale prices to be utilized depending on the initialization of the protocol. The Veridise security analysts also identified 2 medium-severity issues, including an API that updates all listed prices that appears to be broken if more than one price exists and a significant centralization risk as admins can update prices which could spell disaster if admin or feeder keys were stolen. The review additionally identified 6 low issues, which included missing validation in several privileged functions, as well as 4 warnings.

Recommendations. After auditing the protocol, the security analysts had a few suggestions to improve the Decentralized Oracle Integration.

Data Validation. A number of issues identified during this audit corresponded to data validation issues (V-NOR-VUL-005, V-NOR-VUL-006, V-NOR-VUL-007, V-NOR-VUL-008, V-NOR-VUL-011, V-NOR-VUL-012, V-NOR-VUL-013). We would recommend that the developers review their admin functions and, where appropriate, validate that provided arguments are within anticipated bounds. Doing so could prevent potential mistakes when maintaining the protocol.

Centralization. As the oracle retained the ability to act as a centralized oracle, there is a large centralization risk. The Navi developers explained to the Veridise security analysts that this is due to the fact that both decentralized oracle providers have relatively new infrastructure on SUI so they want to maintain the centralized oracle as a fallback in the event of an outage. While we believe this is a valid reason to maintain the centralized features, these features also provide substantial risk if the private keys are stole. We therefore recommend that the Navi developers ensure their key storage mechanism is secure and as few people as possible have access to them. Additionally, we would recommend creating a plan for migration to a fully decentralized oracle at some point in the future.

2 Project Dashboard

Table 2.1: Application Summary.

Name	Version	Type	Platform
Decentralized Oracle Integration	`cb7ea49`	SUI Move	SUI

Table 2.2: Engagement Summary.

Dates	Method	Consultants Engaged	Level of Effort
June 7 - June 13, 2024	Manual & Tools	3	3 person-weeks

Table 2.3: Vulnerability Summary.

Name	Number	Fixed	Acknowledged
Critical-Severity Issues	0	0	0
High-Severity Issues	1	0	1
Medium-Severity Issues	2	1	2
Low-Severity Issues	6	3	5
Warning-Severity Issues	4	1	3
Informational-Severity Issues	0	0	0
TOTAL	13	5	11

Table 2.4: Category Breakdown.

Name	Number
Data Validation	6
Price Manipulation	2
Logic Error	2
Access Control	1
Best Practices	1
Maintainability	1

3 Audit Goals and Scope

3.1 Audit Goals

The engagement was scoped to provide a security assessment of Navi's Decentralized Oracle Integration smart contracts. In our audit, we sought to answer questions such as:

- Can a stale price be used or stored?
- Can a price for an incorrect asset be used or stored?
- Does the oracle provide sufficient availability?
- Are discrepancies between prices provided by different oracle providers resolved?
- Given how relatively new the used oracles are, is monitoring in place to identify potential issues?

3.2 Audit Methodology & Scope

Audit Methodology. To address the questions above, our audit involved a combination of human experts and automated program analysis & testing tools. In particular, we conducted our audit with the aid of the following techniques:

- *Property-based Testing.* To identify some behavioral inconsistencies, we used property-based testing using SUI's build-in testing framework. To do so, we formalized properties that we should hold in the protocol and wrote tests to pseudo-randomly exercise the protocol and check these properties.

Scope. The scope of this audit is limited to the oracle directory of the source code provided by the Navi developers, which contains the smart contract implementation of the Decentralized Oracle Integration.

Methodology. Veridise auditors inspected the provided tests, and read the Decentralized Oracle Integration documentation. They then began a manual audit of the code assisted by SUI's testing infrastructure. During the audit, the Veridise auditors met with the Navi developers to ask questions about the code.

3.3 Classification of Vulnerabilities

When Veridise auditors discover a possible security vulnerability, they must estimate its severity by weighing its potential impact against the likelihood that a problem will arise. Table 3.1 shows how our auditors weigh this information to estimate the severity of a given issue.

In this case, we judge the likelihood of a vulnerability as follows in Table 3.2:

In addition, we judge the impact of a vulnerability as follows in Table 3.3:

Table 3.1: Severity Breakdown.

	Somewhat Bad	Bad	Very Bad	Protocol Breaking
Not Likely	Info	Warning	Low	Medium
Likely	Warning	Low	Medium	High
Very Likely	Low	Medium	High	Critical

Table 3.2: Likelihood Breakdown

Not Likely	A small set of users must make a specific mistake
Likely	Requires a complex series of steps by almost any user(s) - OR - Requires a small set of users to perform an action
Very Likely	Can be easily performed by almost anyone

Table 3.3: Impact Breakdown

Somewhat Bad	Inconveniences a small number of users and can be fixed by the user
Bad	Affects a large number of people and can be fixed by the user - OR - Affects a very small number of people and requires aid to fix
Very Bad	Affects a large number of people and requires aid to fix - OR - Disrupts the intended behavior of the protocol for a small group of users through no fault of their own
Protocol Breaking	Disrupts the intended behavior of the protocol for a large group of users through no fault of their own

4 Vulnerability Report

In this section, we describe the vulnerabilities found during our audit. For each issue found, we log the type of the issue, its severity, location in the code base, and its current status (i.e., acknowledged, fixed, etc.). Table 4.1 summarizes the issues discovered:

Table 4.1: Summary of Discovered Vulnerabilities.

ID	Description	Severity	Status
V-NOR-VUL-001	Staleness of oracle price can exceed expected b...	High	Acknowledged
V-NOR-VUL-002	Cannot update all prices via API	Medium	Fixed
V-NOR-VUL-003	Centralization Risk	Medium	Acknowledged
V-NOR-VUL-004	Users can update prices in their favor	Low	Acknowledged
V-NOR-VUL-005	No validation when changing oracle id	Low	Fixed
V-NOR-VUL-006	No check that oracle is not both primary and se...	Low	Intended Behavior
V-NOR-VUL-007	No validation when updating Oracle's update_int.	Low	Fixed
V-NOR-VUL-008	No check if cast may overflow	Low	Fixed
V-NOR-VUL-009	Changes to update_interval can introduce unfair..	Low	Acknowledged
V-NOR-VUL-010	Pyth price confidence not considered	Warning	Acknowledged
V-NOR-VUL-011	Little validation performed on price feed creation	Warning	Acknowledged
V-NOR-VUL-012	No decimal validation on price update	Warning	Intended Behavior
V-NOR-VUL-013	No validation of decimals on conversion	Warning	Fixed

4.1 Detailed Description of Issues

4.1.1 V-NOR-VUL-001: Staleness of oracle price can exceed expected bounds

Severity	High	**Commit**	cb7ea49
Type	Price Manipulation	**Status**	Acknowledged
File(s)	`oracle.move`		
Location(s)	update_price		
Confirmed Fix At	N/A		

The prices fetched from `OracleProvider` are pushed to the price feeds in the oracle. The oracle considers the last pushed price only valid for `PriceOracle.update_interval` time after the push. This check is performed when the price value is accessed in `oracle::get_token_price()` as shown below.

```
public fun get_token_price(
    clock: &Clock,
    price_oracle: &PriceOracle,
    oracle_id: u8
): (bool, u256, u8) {
    version_verification(price_oracle);

    let price_oracles = &price_oracle.price_oracles;
    assert!(table::contains(price_oracles, oracle_id), error::non_existent_oracle());

    let token_price = table::borrow(price_oracles, oracle_id);
    let current_ts = clock::timestamp_ms(clock);

    let valid = false;
    if (token_price.value > 0 && current_ts - token_price.timestamp <= price_oracle.
    update_interval) {
        valid = true;
    };
    (valid, token_price.value, token_price.decimal)
}
```

Snippet 4.1: Snippet from `get_token_price()`

The timestamp for the last update to the protocol is recorded in the function `oracle::update_price()` . The timestamp for the last update to `PriceOracle` is recorded in the `PriceInfo.timestamp` field for the corresponding price oracle. The timestamp recorded is the time of the update.

Additionally, the price form the `OracleProvider` is considered fresh only for `PriceFeed.max_timestamp_diff` time after the timestamp returned by the downstream decentralized oracles.

The timestamp recorded for the `PriceOracle` is the time of the update and not the timestamp reported from `OracleProvider`.

Impact If an attacker does not update the oracle before initiating the transaction, the oracle may report a price that is stale by `PriceFeed.max_timestamp_diff + PriceOracle.update_interval` rather than by `PriceOracle.update_interval`.

```
public(friend) fun update_price(clock: &Clock, price_oracle: &mut PriceOracle,
    oracle_id: u8, token_price: u256) {
    // TODO: update_token_price can be merged into update_price
    version_verification(price_oracle);

    let price_oracles = &mut price_oracle.price_oracles;
    assert!(table::contains(price_oracles, oracle_id), error::non_existent_oracle());

    let price = table::borrow_mut(price_oracles, oracle_id);
    let now = clock::timestamp_ms(clock);
    //@audit no guarantee that decimals are the same? would it make more sense to
    update decimals here aswell
    // even if they are the same?
    emit(PriceUpdated {
        price_oracle: object::uid_to_address(&price_oracle.id),
        id: oracle_id,
        price: token_price,
        last_price: price.value,
        update_at: now,
        last_update_at: price.timestamp,
    });

    price.value = token_price;
    price.timestamp = now;
}
```

Snippet 4.2: Snippet from `is_oracle_price_fresh()`

```
public fun is_oracle_price_fresh(current_timestamp: u64, oracle_timestamp: u64,
    max_timestamp_diff: u64): bool {
    if (current_timestamp < oracle_timestamp) {
        return false
    };

    return (current_timestamp - oracle_timestamp) < max_timestamp_diff
}
```

Snippet 4.3: Snippet from `is_oracle_price_fresh()`

Recommendation Set the `PriceInfo.timestamp` to the timestamp of the last update from the `OracleProvider`

Developer Response We acknowledge this potential risk. The scenario we are considering is if an actual price change over 25 percent in 60 seconds without recovering in a short amount of time. If nobody updates the oracle price, an attack could use this price difference to create bad debt. However, it is very difficult to address an issue like this as the oracle will need to allow some delay. Under the consideration of timeliness and increase in complexity, we will set a shorter `update_interval` (15 s) to make such an attack less likely to happen.

4.1.2 V-NOR-VUL-002: Cannot update all prices via API

Severity	Medium	**Commit**	cb7ea49
Type	Logic Error	**Status**	Fixed
File(s)	`oracle_pro.move`		
Location(s)	update_prices		
Confirmed Fix At	6b89d4f		

The protocol defines the `update_prices` function to update the prices of all tokens registered with the protocol, as shown below. To do so, it passes in a `PriceInfoObject` object to update the prices provided by the Pyth oracle. This object, however, only contains the price from a single price feed and so it cannot be used to update all of the provided prices. Since `update_single_price` validates that the price feed for a given oracle is correct, this function will therefore always revert as long as there are at least two assets registered.

```
public fun update_prices(
    clock: &Clock,
    oracle_config: &mut OracleConfig,
    price_oracle: &mut PriceOracle,
    supra_oracle_holder: &OracleHolder,
    pyth_state: &State,
    pyth_price_info: &PriceInfoObject
) {
    let feeds = config::get_vec_feeds(oracle_config);
    let len = vector::length(&feeds);
    let i = 0;
    while (i < len) {
        let feed_id = vector::borrow(&feeds, i);
        update_single_price(clock, oracle_config, price_oracle, supra_oracle_holder,
    pyth_state, pyth_price_info, *feed_id);
        i = i + 1;
    };
}
```

Snippet 4.4: Definition of the `update_prices()`

Impact Rather than updating all prices, users will instead have to update the prices of assets individually. This can cause usability issues as users will need to know which prices will be accessed by their transaction and also further incentivizes users to only update prices if it is in their favor (see V-NOR-VUL-004).

Recommendation Consider either removing this function so as to not confuse users if it does not work or allow an array of `PriceInfoObject` to be provided

4.1.3 V-NOR-VUL-003: Centralization Risk

Severity	Medium	**Commit**	cb7ea49
Type	Access Control	**Status**	Acknowledged
File(s)	`oracle.move`		
Location(s)	update_token_price()		
Confirmed Fix At	N/A		

Similar to many projects, Navi OraclePro, the `oracle` module declares an `OracleAdminCap` role that is given special privileges. In particular, these administrators are given the following abilities:

- Grant `OracleFeederCap` capability
- Modify primary and secondary `OracleProviders`.

Specifically, the `OracleFeederCap` allows the owner to update the prices of tokens to arbitrary values using the function `oracle::update_token_price()`

```
public entry fun update_token_price(
    _: &OracleFeederCap,
    clock: &Clock,
    price_oracle: &mut PriceOracle,
    oracle_id: u8,
    token_price: u256,
) {
    version_verification(price_oracle);

    let price_oracles = &mut price_oracle.price_oracles;
    assert!(table::contains(price_oracles, oracle_id), error::non_existent_oracle());
    let price = table::borrow_mut(price_oracles, oracle_id);
    price.value = token_price;
    price.timestamp = clock::timestamp_ms(clock);
}
```

Snippet 4.5: Function `oracle::update_token_price()` from oracle.move

Impact If a private key were stolen, a hacker would have access to sensitive functionality that could compromise the project. For example, an attacker can inflate his collateral by pushing frivolous prices for his collateral tokens. Then the attacker can borrow tokens without enough collaterals.

Recommendation

- As these are all particularly sensitive operations, we would encourage the developers to utilize a decentralized governance or multi-sig contract as opposed to a single account, which introduces a single point of failure.
- Remove the `oracle::update_token_price()` function from `oracle.move()`

Developer Response We will keep the interface for a short time after the official upgrade and then remove it.

4.1.4 V-NOR-VUL-004: Users can update prices in their favor

Severity	Low	Commit	cb7ea49
Type	Price Manipulation	Status	Acknowledged
File(s)	`oracle_pro.move`		
Location(s)	update_single_price		
Confirmed Fix At	N/A		

The oracle module relies on users to push prices to the oracle module as they use the protocol. They are not require to do so, however, unless the current price has expired. As asset prices can be updated individually, users may selectively update prices however they wish.

```
public fun get_token_price(
    clock: &Clock,
    price_oracle: &PriceOracle,
    oracle_id: u8
): (bool, u256, u8) {
    version_verification(price_oracle);

    let price_oracles = &price_oracle.price_oracles;
    assert!(table::contains(price_oracles, oracle_id), error::non_existent_oracle());

    let token_price = table::borrow(price_oracles, oracle_id);
    let current_ts = clock::timestamp_ms(clock);

    let valid = false;
    if (token_price.value > 0 && current_ts - token_price.timestamp <= price_oracle.
    update_interval) {
        valid = true;
    };
    (valid, token_price.value, token_price.decimal)
}
```

Snippet 4.6: Definition of the `get_token_price()` function

Impact By monitoring asset prices, users can ensure that prices are updated in their favor, potentially allowing unfavorable transactions to be processed if the update interval is long enough.

Recommendation Either maintain a small `update_interval` or pull prices as necessary.

Developer Response We believe a 30 second interval in addition to all user price updates from the front-end will make this type of manipulation unlikely. We will, however, monitor the protocol to determine if it occurs in practice.

4.1.5 V-NOR-VUL-005: No validation when changing oracle id

Severity	Low	**Commit**	cb7ea49
Type	Data Validation	**Status**	Fixed
File(s)	`config.move`		
Location(s)	set_oracle_id_to_price_feed		
Confirmed Fix At	f7e6926		

The oracle module uses a numeric identifier as a method of identifying price feeds in the protocol. While this identifier should never need to be changed, they do provide a function for the admin to do so if deemed necessary. However, the function provided is missing a check performed during the creation of a price feed that determines if there is an existing feed with the same identifier.

```
1 public(friend) fun set_oracle_id_to_price_feed(cfg: &mut OracleConfig, feed_id:
      address, value: u8) {
2     assert!(table::contains(&cfg.feeds, feed_id), error::price_feed_not_found());
3     let price_feed = table::borrow_mut(&mut cfg.feeds, feed_id);
4     let before_value = price_feed.oracle_id;
5
6     price_feed.oracle_id = value;
7     emit(PriceFeedSetOracleId {config: object::uid_to_address(&cfg.id), feed_id:
      feed_id, value: value, before_value: before_value})
8 }
```

Snippet 4.7: Definition of the `set_oracle_id_to_price_feed` function which is missing a check for duplicate ID

Impact As price feeds are fetched using oracle IDs, duplicate IDs could allow prices to accidentally be fetched for a different asset than what is expected.

Recommendation Perform the same duplication check that is performed when a price feed is created.

4.1.6 V-NOR-VUL-006: No check that oracle is not both primary and secondary

Severity	Low	Commit	cb7ea49
Type	Data Validation	Status	Intended Behavior
File(s)	config.move		
Location(s)	set_primary_oracle_provider, set_secondary_oracle_provider		
Confirmed Fix At	N/A		

The oracle module allows prices to be provided by a primary provider and a secondary provider. If two providers are available, Navi will attempt to ensure better availability of the oracle by setting prices from the secondary provider if the primary is not available. Additionally, if prices are available from both providers, Navi will check the reliability of the prices by performing additional sanity checks to ensure that the prices match within a threshold of error. When setting the primary and secondary provider, however, they do not check to ensure that the same provider is not used as both primary and secondary.

```
public(friend) fun set_primary_oracle_provider(cfg: &mut OracleConfig, feed_id:
    address, provider: OracleProvider) {
    assert!(table::contains(&cfg.feeds, feed_id), error::price_feed_not_found());
    let price_feed = table::borrow_mut(&mut cfg.feeds, feed_id);
    if (price_feed.primary == provider) {
        return
    };
    let before_provider = price_feed.primary;

    assert!(table::contains(&price_feed.oracle_provider_configs, provider), error::
    provider_config_not_found());
    let provider_config = table::borrow(&price_feed.oracle_provider_configs, provider
    );
    assert!(oracle_provider::is_oracle_provider_config_enable(provider_config), error
    ::oracle_provider_disabled());
    price_feed.primary = provider;

    emit(SetOracleProvider {config: object::uid_to_address(&cfg.id), feed_id: feed_id
    , is_primary: true, provider: oracle_provider::to_string(&provider),
    before_provider: oracle_provider::to_string(&before_provider)});
}
```

Snippet 4.8: Definition of the `set_primary_oracle_provider` function which does not check the secondary provider

Impact If the same oracle provider was both the primary and secondary, the protocol would not benefit from better availability or reliability. Rather, it can create a false sense of security that the oracle is working as intended.

Recommendation Add some additional checks if the new provider is already used as primary or secondary.

Developer Response The potential benefit of allowing the same provider to be primary and secondary is to allow swaps without a third or empty provider. However, we will closely check

the provider configurations to avoid the false sense of security as mentioned.

4.1.7 V-NOR-VUL-007: No validation when updating Oracle's update_interval

Severity	Low	Commit	cb7ea49
Type	Data Validation	Status	Fixed
File(s)	oracle.move		
Location(s)	set_update_interval()		
Confirmed Fix At	7effbe2		

The protocol considers the prices pushed to its feeds valid only for `PriceOracle.update_interval` amount of time after the last update to the price stored in the `PriceInfo.timestamp`. This check is performed in `get_token_price()` when a token price is accessed.

```
public fun get_token_price(
    clock: &Clock,
    price_oracle: &PriceOracle,
    oracle_id: u8
): (bool, u256, u8) {
    version_verification(price_oracle);

    let price_oracles = &price_oracle.price_oracles;
    assert!(table::contains(price_oracles, oracle_id), error::non_existent_oracle());

    let token_price = table::borrow(price_oracles, oracle_id);
    let current_ts = clock::timestamp_ms(clock);

    let valid = false;
    if (token_price.value > 0 && current_ts - token_price.timestamp <= price_oracle.
    update_interval) {
        valid = true;
    };
    (valid, token_price.value, token_price.decimal)
}
```

Snippet 4.9: Definition of `get_token_price()`

A user with `OracleAdminCap` capability can update the value of the parameter `PriceOracle.update_interval` in the function `oracle::set_update_interval()`

```
public entry fun set_update_interval(
    _: &OracleAdminCap,
    price_oracle: &mut PriceOracle,
    update_interval: u64,
) {
    version_verification(price_oracle);

    price_oracle.update_interval = update_interval;
}
```

Snippet 4.10: Snippet from `oracle::set_update_interval()`

This function is missing validation on the value of the argument `update_interval`.

Impact If the new value of `update_interval` is zero, the oracle will be rendered useless until the value is set to be non-zero.

If this parameter is set to a very small value, the oracle prices will become stale too often

If this parameter is set to a very large value, the oracle may return stale values causing losses to protocol or users.

Recommendation Add bounds check on the value of the `update_interval` argument to the function `oracle::set_update_interval()`

4.1.8 V-NOR-VUL-008: No check if cast may overflow

Severity	Low	**Commit**	cb7ea49
Type	Data Validation	**Status**	Fixed
File(s)	`strategy.move`		
Location(s)	validate_price_difference		
Confirmed Fix At	3285cde		

The function `validate_price_difference` checks how reliable the price data provided by the oracles is. To do this, it calculates the difference between two prices using the `utils::calculate_amplitude()` function, which returns a `u256` value. This value is then casted to `u64` at the call site without any checks, which can cause an overflow if the prices are significantly different.

```
let diff = (utils::calculate_amplitude(primary_price, secondary_price) as u64);

if (diff < threshold1) { return constants::level_normal() };
if (diff > threshold2) { return constants::level_critical() };
```

Snippet 4.11: Snippet from `validate_price_difference()`

Impact If the difference between two prices is significant, this can cause the cast to overflow and prevent the Navi-protocol administrators from receiving an important `PriceRegulation` event that would normally alert them to a price anomaly.

Recommendation It is recommended to implement a check that, in case of a potential overflow, would return the `constants::level_critical()` result.

4.1.9 V-NOR-VUL-009: Changes to update_interval can introduce unfairness

Severity	Low	**Commit**	cb7ea49
Type	Logic Error	**Status**	Acknowledged
File(s)	oracle::set_update_interval		
Location(s)	oracle.move		
Confirmed Fix At	N/A		

The function `oracle::set_update_interval` allows the admin to change the `update_interval` value that influences the validity of token prices returned by `oracle::get_token_price`.

Consider the following scenario:

- A user tries to retrieve the latest token price, and if the price is considered outdated, they receive a value with the `valid` flag set to `false`.
- Based on this information, the user decides to take an action (or not).
- An administrator changes the `update_interval` to a higher value. Now, when the user makes the same request to `oracle::get_token_price`, they will receive a valid price.

This scenario can lead to unfairness, as the price that one user considers outdated may be considered valid for another user. This can result in unexpected losses for some users.

Impact The ability of the admin to arbitrarily change `update_interval` introduces unfairness among users.

Recommendation It is recommended to change `update_interval` value only when all prices are considered fresh.

Developer Response The risk of this issue is relatively low and this check may decrease our flexibility to address unforeseen issues. For example, if something happens to our oracle and we need to stop it then set the interval. Since it is stopped, all prices get stale and we are not able to set the interval.

4.1.10 V-NOR-VUL-010: Pyth price confidence not considered

Severity	Warning	**Commit**	cb7ea49
Type	Best Practices	**Status**	Acknowledged
File(s)	`adaptor_pyth.move`		
Location(s)	get_price_native		
Confirmed Fix At	N/A		

In addition to the price of an asset, Pyth reports a confidence interval which indicates a lower bound and upper bound on the actual price of the asset. In Pyth's best practices documentation, they recommend accounting for these intervals when determining the price but they are not used by the Oracle module as shown below.

```
public fun get_price_native(clock: &Clock, pyth_state: &State, pyth_price_info: &
    PriceInfoObject): (u64, u64, u64){
    let pyth_price_info = pyth::get_price(pyth_state, pyth_price_info, clock);

    let i64_price = price::get_price(&pyth_price_info);
    let i64_expo = price::get_expo(&pyth_price_info);
    let timestamp = price::get_timestamp(&pyth_price_info) * 1000; // timestamp from
    pyth in seconds, should be multiplied by 1000
    let price = i64::get_magnitude_if_positive(&i64_price);
    let expo = i64::get_magnitude_if_negative(&i64_expo);

    (price, expo, timestamp)
}
```

Snippet 4.12: Snippet from `example()`

Impact Pyth recommends using the confidence interval to protect against unusual market conditions.

Recommendation Consider reducing the price by the confidence interval when determining the price of collateral and increasing the price by the interval when determining he price of borrowed assets.

Developer Response We use 2 providers to guarantee the price which is an alternative confidence evaluation method. Additionally, the protocol's health factors should protect against price fluctuations in a similar way to the confidence interval.

4.1.11 V-NOR-VUL-011: Little validation performed on price feed creation

Severity	Warning	**Commit**	cb7ea49
Type	Data Validation	**Status**	Acknowledged
File(s)	`config.move`		
Location(s)	new_price_feed		
Confirmed Fix At	N/A		

To add a new price feed to the oracle, an admin must must call `new_price_feed` to create a PriceFeed object. This object contains information about the price feed such as the primary and secondary providers, but also safety information that is used when performing sanity checks between the primary and secondary oracle.

```
public(friend) fun new_price_feed<CoinType>(
        cfg: &mut OracleConfig,
        oracle_id: u8,
        max_timestamp_diff: u64,
        price_diff_threshold1: u64,
        price_diff_threshold2: u64,
        max_duration_within_thresholds: u64,
        maximum_allowed_span_percentage: u64,
        maximum_effective_price: u256,
        minimum_effective_price: u256,
        historical_price_ttl: u64,
        ctx: &mut TxContext,
    ) {
        assert!(!is_price_feed_exists<CoinType>(cfg, oracle_id), error::
    price_feed_already_exists());

        let uid = object::new(ctx);
        let object_address = object::uid_to_address(&uid);
        let feed = PriceFeed {
            ...
        };

        table::add(&mut cfg.feeds, object_address, feed);
        vector::push_back(&mut cfg.vec_feeds, object_address);

        emit(PriceFeedCreated {sender: tx_context::sender(ctx), config: object::
    uid_to_address(&cfg.id), feed_id: object_address})
    }
```

Snippet 4.13: Definition of the `new_price_feed()` function

Impact If these parameters are set incorrectly, it could cause these safety checks to be irrelevant or could prevent the oracle from successfully fetching a price. As an example if `price_diff_threshold1` and `price_diff_threshold2` are both set to 0, prices will only be set when the primary and secondary providers return the exact same price (assuming both are available).

Recommendation Add additional validation to prevent mistakes when configuring price feeds. Similar validation should be added to the setter functions as well.

4.1.12 V-NOR-VUL-012: No decimal validation on price update

Severity	Warning	**Commit**	cb7ea49
Type	Maintainability	**Status**	Intended Behavior
File(s)	`oracle.move`		
Location(s)	update_price, update_token_price		
Confirmed Fix At	N/A		

The oracle module allows prices to be updated from external oracle providers. After calculating the new price, the `update_price` function will be invoked to set the new price in the oracle. When updating the token price, the number of decimals is not passed to the function, as shown below, because it is assumed that the price has already been converted to the number of decimals used by the oracle. As new oracle providers may be added in the future, we would still recommend validating the price's decimals to prevent any future errors.

```
public(friend) fun update_price(
    clock: &Clock,
    price_oracle: &mut PriceOracle,
    oracle_id: u8,
    token_price: u256
) {
    ...
}
```

Snippet 4.14: Definition of the `update_price()` function

Impact Should a price with the wrong number of decimals be passed to the oracle, it could cause assets to be drastically under- or over-priced.

Recommendation Validate the price's decimals in both `update_price` and `update_token_price` .

Developer Response We convert the decimal from provider side to our internal config digit via `get_price_from_adaptor` for current or future providers.

4.1.13 V-NOR-VUL-013: No validation of decimals on conversion

Severity	Warning	**Commit**	cb7ea49
Type	Data Validation	**Status**	Fixed
File(s)	`oracle_utils.move`		
Location(s)	to_target_decimal_value		
Confirmed Fix At	b91b1a4		

The oracle works with prices that are represented as integers with an additional decimal value that indicates the number of decimal places allocated for the fraction of the number. The maximum number of decimal places that can be represented is `78` as the price is stored as a `u256` value. Also, the actual number of decimals should be much less than that, since there has to be room for the integer part of the price as well.

```
public fun to_target_decimal_value(value: u256, decimal: u8, target_decimal: u8):
    u256 {
        assert!(decimal > 0 && target_decimal > 0, 1);
        while (decimal != target_decimal) {
            if (decimal < target_decimal) {
                value = value * 10;
                decimal = decimal + 1;
            } else {
                value = value / 10;
                decimal = decimal - 1;
            };
        };

        value
    }
```

Snippet 4.15: Snippet from `to_target_decimal_value()`

Impact If the `decimal` value of the `Price` is incorrectly chosen, the protocol will not be able to process price updates because of an integer overflow in the `to_target_decimal_value` function, which is called each time a new price is sent to the oracle.

Recommendation It is recommended to limit the decimal value of the price to no more than 78.