Zellic

Prepared for
Aditya Arora
Pyth Network

Prepared by
Katerina Belotskaia
Kritsada Dechawattana
Zellic

July 29, 2025

Pyth Morpho Wrapper

Smart Contract Security Assessment

Contents

About Zellic

Zellic is a vulnerability research firm with deep expertise in blockchain security. We specialize in EVM, Move (Aptos and Sui), and Solana as well as Cairo, NEAR, and Cosmos. We review L1s and L2s, cross-chain protocols, wallets and applied cryptography, zero-knowledge circuits, web applications, and more.

Prior to Zellic, we founded the #1 CTF (competitive hacking) team ↗ worldwide in 2020, 2021, and 2023. Our engineers bring a rich set of skills and backgrounds, including cryptography, web security, mobile security, low-level exploitation, and finance. Our background in traditional information security and competitive hacking has enabled us to consistently discover hidden vulnerabilities and develop novel security research, earning us the reputation as the go-to security firm for teams whose rate of innovation outpaces the existing security landscape.

For more on Zellic's ongoing security research initiatives, check out our website zellic.io ↗ and follow @zellic_io ↗ on Twitter. If you are interested in partnering with Zellic, contact us at hello@zellic.io ↗.

1. Overview

1.1. Executive Summary

Zellic conducted a security assessment for Pyth Network from July 8th to July 9th, 2025. During this engagement, Zellic reviewed Pyth Morpho Wrapper's code for security vulnerabilities, design issues, and general weaknesses in security posture.

1.2. Goals of the Assessment

In a security assessment, goals are framed in terms of questions that we wish to answer. These questions are agreed upon through close communication between Zellic and the client. In this assessment, we sought to answer the following questions:

- Can a misconfigured oracle or vault setup lead to incorrect price reporting?
- Is the feed-age logic correctly enforced, and is it resistant to stale data?
- Can an attacker game the vault-price sampling to skew the oracle output?
- Are there any unbounded external calls or dynamic loops that may introduce gas-griefing risks?

1.3. Non-goals and Limitations

We did not assess the following areas that were outside the scope of this engagement:

- Front-end components
- Infrastructure relating to the project
- Key custody

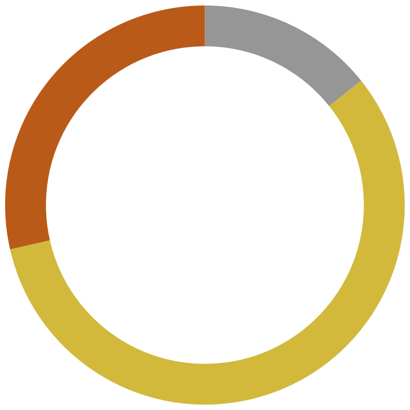
Due to the time-boxed nature of security assessments in general, there are limitations in the coverage an assessment can provide.

1.4. Results

During our assessment on the scoped Pyth Morpho Wrapper contracts, we discovered seven findings. No critical issues were found. Two findings were of high impact, four were of medium impact, and the remaining finding was informational in nature.

Breakdown of Finding Impacts

Impact Level	Count
🟥 Critical	0
🟧 High	2
🟨 Medium	4
🟩 Low	0
⬜ Informational	1



2. Introduction

2.1. About Pyth Morpho Wrapper

Pyth Network contributed the following description of Pyth Morpho Wrapper:

> Pyth Morpho Wrapper that connect wraps Pyth oracles with morpho's IOracle interface.

2.2. Methodology

During a security assessment, Zellic works through standard phases of security auditing, including both automated testing and manual review. These processes can vary significantly per engagement, but the majority of the time is spent on a thorough manual review of the entire scope.

Alongside a variety of tools and analyzers used on an as-needed basis, Zellic focuses primarily on the following classes of security and reliability issues:

> **Basic coding mistakes.** Many critical vulnerabilities in the past have been caused by simple, surface-level mistakes that could have easily been caught ahead of time by code review. Depending on the engagement, we may also employ sophisticated analyzers such as model checkers, theorem provers, fuzzers, and so on as necessary. We also perform a cursory review of the code to familiarize ourselves with the contracts.
>
> **Business logic errors.** Business logic is the heart of any smart contract application. We examine the specifications and designs for inconsistencies, flaws, and weaknesses that create opportunities for abuse. For example, these include problems like unrealistic tokenomics or dangerous arbitrage opportunities. To the best of our abilities, time permitting, we also review the contract logic to ensure that the code implements the expected functionality as specified in the platform's design documents.
>
> **Integration risks.** Several well-known exploits have not been the result of any bug within the contract itself; rather, they are an unintended consequence of the contract's interaction with the broader DeFi ecosystem. Time permitting, we review external interactions and summarize the associated risks: for example, flash loan attacks, oracle price manipulation, MEV/sandwich attacks, and so on.
>
> **Code maturity.** We look for potential improvements in the codebase in general. We look for violations of industry best practices and guidelines and code quality standards. We also provide suggestions for possible optimizations, such as gas optimization, upgradability weaknesses, centralization risks, and so on.

For each finding, Zellic assigns it an impact rating based on its severity and likelihood. There is no hard-and-fast formula for calculating a finding's impact. Instead, we assign it on a case-by-case basis based on our judgment and experience. Both the severity and likelihood of an issue affect its impact. For instance, a highly severe issue's impact may be attenuated by a low likelihood. We assign the following impact ratings (ordered by importance): Critical, High, Medium, Low, and

Informational.

Zellic organizes its reports such that the most important findings come first in the document, rather than being strictly ordered on impact alone. Thus, we may sometimes emphasize an "Informational" finding higher than a "Low" finding. The key distinction is that although certain findings may have the same impact rating, their *importance* may differ. This varies based on various soft factors, like our clients' threat models, their business needs, and so on. We aim to provide useful and actionable advice to our partners considering their long-term goals, rather than a simple list of security issues at present.

2.3. Scope

The engagement involved a review of the following targets:

Pyth Morpho Wrapper Contracts

Type	Solidity
Platform	EVM-compatible

Target	pyth-morpho-wrapper
Repository	https://github.com/pyth-network/pyth-morpho-wrapper ↗
Version	`89091d9589585817667f879a5c4ede025ca3bdab`

Programs	`libraries/PythErrorsLib`
	`libraries/PythFeedLib`
	`libraries/VaultLib`
	`MorphoPythOracle`
	`MorphoPythOracleFactory`

2.4. Project Overview

Zellic was contracted to perform a security assessment for a total of 0.5 person-weeks. The assessment was conducted by two consultants over the course of two calendar days.

Contact Information

The following project managers were associated with the engagement:

Jacob Goreski
Engagement Manager
jacob@zellic.io ↗

Chad McDonald
Engagement Manager
chad@zellic.io ↗

The following consultants were engaged to conduct the assessment:

Katerina Belotskaia
Engineer
kate@zellic.io ↗

Kritsada Dechawattana
Engineer
kritsada@zellic.io ↗

2.5. Project Timeline

The key dates of the engagement are detailed below.

July 8, 2025	Kick-off call
July 8, 2025	Start of primary review period
July 9, 2025	End of primary review period

3. Detailed Findings

3.1. Inflexible price staleness check for diverse asset types

Target	MorphoPythOracle		
Category	Business Logic	**Severity**	High
Likelihood	Medium	**Impact**	High

Description

The MorphoPythOracle contract accepts the `priceFeedMaxAge` that is used to prevent a stale price for all feeds. However, the oracle currently uses a single `priceFeedMaxAge` for all feeds. This can lead to good prices from slow-moving assets being rejected or stale prices from fast-moving feeds being accepted.

For example, fast-moving assets may need `priceFeedMaxAge` = 15s, while USDC might be accepted with `priceFeedMaxAge` = 60s. Using the same staleness check for all assets (e.g., always <= 30s) may reject good USDC prices just because they have not changed recently or accept stale prices for fast-moving assets like MEME coins.

```
// MorphoPythOracle.sol

constructor(
    address pyth_,
    IERC4626 baseVault,
    uint256 baseVaultConversionSample,
    bytes32 baseFeed1,
    bytes32 baseFeed2,
    uint256 baseTokenDecimals,
    IERC4626 quoteVault,
    uint256 quoteVaultConversionSample,
    bytes32 quoteFeed1,
    bytes32 quoteFeed2,
    uint256 quoteTokenDecimals,
@>  uint256 priceFeedMaxAge
) {
    [...]

@>  PRICE_FEED_MAX_AGE = priceFeedMaxAge;
}
```

```
// VaultLib.sol
```

```
function getPrice(IPyth pyth, bytes32 priceId, uint256 maxAge)
    internal view returns (uint256) {
    if (priceId == bytes32(0)) return 1;

    PythStructs.Price memory price = pyth.getPriceNoOlderThan(priceId,
    maxAge);
    require(int256(price.price) >= 0, PythErrorsLib.NEGATIVE_ANSWER);
    return uint256(int256(price.price));
}
```

Impact

Since only one `priceFeedMaxAge` is currently used for all price feeds, it could lead to serious issues with the prices returned from the Pyth contract. Different tokens have different heartbeats, so a single universal `priceFeedMaxAge` could be either too large or too small. As a result, stale prices might be accepted as valid, or valid ones rejected as stale.

Recommendations

A better approach would be to assign a specific `priceFeedMaxAge` per feed rather than using a single value. This would allow the system to fine-tune staleness tolerances based on the characteristics of each individual feed. For example, high-volatility tokens could have tighter constraints, while stablecoins or slower-moving assets could be permitted longer validity periods. This change would improve both reliability and accuracy of price computation across a wide range of asset types.

Remediation

This issue has been acknowledged by Pyth Network.

Pyth Network provided the following response to this finding:

> Because Pyth is a pull-based oracle, anyone can update the on-chain feeds with the newest data before performing an operation that utilizes the oracle. Even for less volatile assets like USDC, there is an update available roughly every second (no matter how much the price has changed) and anyone can post this on-chain. "Using the same staleness check for all assets (e.g., always <= 30s) may reject good USDC prices just because they have not changed recently" is therefore wrong and we recommend to use the same (low) value for both feeds, for instance 15s in the provided example. Applying the recommendation and using a higher value for historically stable coins would worsen the security of the system.

3.2. Limited test coverage

Target	MorphoPythOracle, MorphoPythOracleFactory		
Category	Code Maturity	**Severity**	High
Likelihood	High	**Impact**	High

Description

The test coverage for the MorphoPythOracle contract is currently minimal and omits a wide range of edge cases and negative scenarios. For the MorphoPythOracleFactory contract, no tests are implemented at all.

Even though this project is a fork of morpho-blue-oracles ↗, full test coverage should still be implemented, since the implementation has changed significantly, particularly due to the integration of Pyth protocol.

Impact

Incomplete testing increases the risk of silent misbehavior, particularly in oracle-price–computation paths. Without coverage for edge values, stale feeds, and misconfiguration cases, bugs or vulnerabilities could go undetected in production.

Examples of missing coverage include the following:

- Vaults specified as nonzero addresses and used in the price calculation
- Handling of omitted vaults or feeds
- Extreme `conversionSample` values that could cause rounding errors
- Reverts due to stale price data
- Proper behavior of factory deployment and duplicate prevention

Recommendations

When building a complex contract ecosystem with multiple moving parts and dependencies, comprehensive testing is essential. This includes testing for both positive and negative scenarios. Positive tests should verify that each function's side effect is as expected, while negative tests should cover every revert, preferably in every logical branch.

The test coverage for this project should be expanded to include all contracts, not just surface-level functions.

Therefore, we recommend building a rigorous test suite that includes all contracts to ensure that the system operates securely and as intended.

Good test coverage has multiple effects.

- It finds bugs and design flaws early (preaudit or prerelease).
- It gives insight into areas for optimization (e.g., gas cost).
- It displays code maturity.
- It bolsters customer trust in your product.
- It improves understanding of how the code functions, integrates, and operates — for developers and auditors alike.
- It increases development velocity long-term.

Remediation

This issue has been acknowledged by Pyth Network.

3.3. Acceptance of untrusted Pyth addresses and invalid feed IDs

Target	MorphoPythOracleFactory		
Category	Coding Mistakes	**Severity**	Medium
Likelihood	Medium	**Impact**	Medium

Description

The `createMorphoPythOracle` function accepts any `pyth` address and any set of feed IDs. If a fake `pyth` address, an inconsistent feed combination, or duplicated IDs are supplied, the oracle still deploys successfully and returns a price.

```
function createMorphoPythOracle(
@>  address pyth,
    IERC4626 baseVault,
    uint256 baseVaultConversionSample,
@>  bytes32 baseFeed1,
@>  bytes32 baseFeed2,
    uint256 baseTokenDecimals,
    IERC4626 quoteVault,
    uint256 quoteVaultConversionSample,
@>  bytes32 quoteFeed1,
@>  bytes32 quoteFeed2,
    uint256 quoteTokenDecimals,
    uint256 priceFeedMaxAge,
    bytes32 salt
) external returns (IMorphoPythOracle oracle) {

    [...]

}
```

Impact

There is risk of price manipulation. A malicious `pyth` address can return manipulated prices, allowing attackers to profit from incorrect exchange rates or affect downstream protocols relying on this oracle.

There is also an increased attack surface. Malicious actors may repeatedly deploy oracles with misleading inputs to confuse or exploit other systems expecting consistent feed behavior.

Recommendations

To mitigate this issue, a possible solution is to implement an allowlist of trusted `pyth` addresses and reject duplicate feed IDs. This also helps catch invalid IDs, since `getPriceUnsafe` will revert during deployment if a `priceId` does not exist.

Remediation

This issue has been acknowledged by Pyth Network.

Pyth Network provided the following response to this finding:

> `MorphoPythOracle` closely follows the design philosophy and trust properties of morpho-blue-oracles ↗, to which the same consideration applies (for the feed addresses). As mentioned in the Morpho risk section ↗ and the documentation ↗, market creators and users need to carefully validate the oracle address and its configuration.

3.4. ERC-4626 vault's exchange-rate manipulation

Target	MorphoPythOracleFactory, MorphoPythOracle		
Category	Protocol Risks	**Severity**	Medium
Likelihood	Medium	**Impact**	Medium

Description

The `createMorphoPythOracle` function of the MorphoPythOracleFactory contract allows a user to specify arbitrary `baseVault` and `quoteVault` ERC-4626 addresses.

In the MorphoPythOracle contract, for every `price()` call, the oracle uses each vault to convert a fixed share amount into underlying assets (`convertToAssets`), multiplies those amounts by Pyth feed prices, and divides the two sides, directly influencing the result. So if an attacker can change the share price, they can shift the oracle price.

```
function price() external view returns (uint256) {
    return SCALE_FACTOR.mulDiv(
        BASE_VAULT.getAssets(BASE_VAULT_CONVERSION_SAMPLE)
            * PythFeedLib.getPrice(pyth, BASE_FEED_1, PRICE_FEED_MAX_AGE)
            * PythFeedLib.getPrice(pyth, BASE_FEED_2, PRICE_FEED_MAX_AGE),
        QUOTE_VAULT.getAssets(QUOTE_VAULT_CONVERSION_SAMPLE)
            * PythFeedLib.getPrice(pyth, QUOTE_FEED_1, PRICE_FEED_MAX_AGE)
            * PythFeedLib.getPrice(pyth, QUOTE_FEED_2, PRICE_FEED_MAX_AGE)
    );
}
```

There are several potential manipulation vectors that can affect `convertToAssets`:

- **Direct donation.** An attacker can transfer underlying tokens directly to the vault, so if `totalAssets` is calculated as the token balance, `totalAssets` rises while `totalSupply` stays the same, so the price per share and therefore the oracle price goes up.
- **First-deposit front-run.** If the vault starts empty, the first depositor can effectively set an arbitrary initial exchange rate.
- **Flash loan.** If the vault supports flash loans, an attacker can momentarily remove or add assets in the same block the oracle is read.
- **Dynamic fees.** If a vault supports dynamic fees, an owner can raise or lower fees, changing the `convertToAssets` result.

Impact

Since an attacker could fully control the vault contracts or influence even trusted vaults to affect the `convertToAssets` result used in price calculation, it could lead to a serious issue, when the attacker can shift the entire oracle price in their favor, causing over-borrowing or forced liquidations in the markets that rely on this oracle.

Recommendations

Consider adding the following protections:

1. Whitelist vaults that can be used during `createMorphoPythOracle` execution, and reject arbitrary addresses.
2. Add a per-block deviation cap, and revert if `convertToAssets` changes by more than `X%` within a single block.
3. Document these constraints so users understand which vaults are acceptable.

Remediation

This issue has been acknowledged by Pyth Network.

Pyth Network provided the following response to this finding:

> MorphoPythOracle closely follows the design philosophy and trust properties of morpho-blue-oracles ↗, to which the same consideration applies. As mentioned in the Morpho risk section ↗ and the documentation ↗, market creators and users need to carefully validate the oracle address and its configuration, including the chosen vault.

3.5. Insufficient conversion sample can break price calculation

Target	MorphoPythOracle		
Category	Coding Mistakes	**Severity**	Medium
Likelihood	Low	**Impact**	Medium

Description

The MorphoPythOracle contract accepts arbitrary values for the `baseVaultConversionSample` and `quoteVaultConversionSample` in its constructor. These values are later used in calls to the vaults' `convertToAssets()` function, which converts shares into asset units. However, when the provided conversion sample is too small, especially in vaults with large share-to-asset ratios or high precision rounding, the result of `convertToAssets(shares)` may round down to zero.

```
// MorphoPythOracle.sol

constructor(
    address pyth_,
    IERC4626 baseVault,
    uint256 baseVaultConversionSample,
    bytes32 baseFeed1,
    bytes32 baseFeed2,
    uint256 baseTokenDecimals,
    IERC4626 quoteVault,
    uint256 quoteVaultConversionSample,
    bytes32 quoteFeed1,
    bytes32 quoteFeed2,
    uint256 quoteTokenDecimals,
    uint256 priceFeedMaxAge
) {
    [...]
@>  BASE_VAULT_CONVERSION_SAMPLE = baseVaultConversionSample;
    QUOTE_VAULT = quoteVault;
@>  QUOTE_VAULT_CONVERSION_SAMPLE = quoteVaultConversionSample;
    [...]
}
```

```
// MorphoPythOracle.sol

function price() external view returns (uint256) {
    return SCALE_FACTOR.mulDiv(
        BASE_VAULT.getAssets(BASE_VAULT_CONVERSION_SAMPLE)
            * PythFeedLib.getPrice(pyth, BASE_FEED_1, PRICE_FEED_MAX_AGE)
            * PythFeedLib.getPrice(pyth, BASE_FEED_2, PRICE_FEED_MAX_AGE),
        QUOTE_VAULT.getAssets(QUOTE_VAULT_CONVERSION_SAMPLE)
            * PythFeedLib.getPrice(pyth, QUOTE_FEED_1, PRICE_FEED_MAX_AGE)
            * PythFeedLib.getPrice(pyth, QUOTE_FEED_2, PRICE_FEED_MAX_AGE)
    );
}
```

```
// VaultLib.sol

function getAssets(IERC4626 vault, uint256 shares)
    internal view returns (uint256) {
    if (address(vault) == address(0)) return 1;

    return vault.convertToAssets(shares);
}
```

Impact

If `BASE_VAULT_CONVERSION_SAMPLE` or `QUOTE_VAULT_CONVERSION_SAMPLE` are configured incorrectly, and the provided amounts aren't large enough for `convertToAssets` to return a value corresponding to one asset unit, the result can be rounded to zero. As a result, this would make the oracle return invalid pricing, which could break integrations or logic that depend on a reliable price feed.

Recommendations

To mitigate this issue, consider enforcing validation at deployment time to ensure that `convertToAssets(sample)` returns a nonzero value for both base and quote. This can be done by calling `convertToAssets()` with the provided samples during the constructor and reverting if either result is zero.

Remediation

This issue has been acknowledged by Pyth Network.

Pyth Network provided the following response to this finding:

> MorphoPythOracle closely follows the design philosophy and trust properties of morpho-blue-oracles ↗, to which the same consideration applies. The Morpho

documentation ↗ points out: "Finding the right balance for vault conversion samples is critical. The samples need to be large enough to provide sufficient precision when converting share values to asset values, but not so large that they cause the scale factor calculation to underflow or become 0 through division." Enforcing the return value to be greater than zero would only handle one specific edge case. It would not remove the responsibility of the oracle creator to choose the parameter appropriately or test the oracle after deployment, which should always be done and is explicitly highlighted in the Morpho documentation.

3.6. Lack of confidence-interval verification

Target	MorphoPythOracle		
Category	Coding Mistakes	**Severity**	Medium
Likelihood	Medium	**Impact**	Medium

Description

The `pyth.getPriceNoOlderThan` function returns a `Price` struct with four fields:

```
struct Price {
    // Price
    int64 price;
    // Confidence interval around the price
    uint64 conf;
    // Price exponent
    int32 expo;
    // Unix timestamp describing when the price was published
    uint publishTime;
}
```

In the `getPrice` function, there is only a check that `price` is nonnegative, and there is no verification of `conf` and `expo`. The `publishTime` is implicitly handled by `getPriceNoOlderThan`, which already enforces the `maxAge` limit.

Impact

The Pyth protocol aggregates prices from multiple providers into a final price and a confidence interval. The confidence interval reflects the variation between provided prices and a large `conf` means the price is highly uncertain. So if the `conf` field is ignored, there's a risk of using an inaccurate price even if it's considered valid based on `publishTime`.

Recommendations

Consider adding confidence-interval verification.

Remediation

This issue has been acknowledged by Pyth Network.

3.7. Lack of NatSpec documentation

Target	MorphoPythOracleFactory, MorphoPythOracle		
Category	Code Maturity	**Severity**	Informational
Likelihood	N/A	**Impact**	Informational

Description

The MorphoPythOracleFactory and MorphoPythOracle contracts expose numerous configurable parameters and functions that influence critical behavior such as price calculation, feed selection, vault interactions, and staleness tolerance. However, the codebase lacks comprehensive NatSpec documentation on key components, including functions, constructor arguments, state variables, and their intended usage.

Impact

The absence of NatSpec increases the likelihood of misconfiguration, integration bugs, and trust issues, particularly in a protocol component as sensitive as a price oracle.

Recommendations

Add comprehensive NatSpec comments to all public/external functions, constructors, and key state variables across the MorphoPythOracle and MorphoPythOracleFactory contracts.

Remediation

This issue has been acknowledged by Pyth Network, and a fix was implemented in commit `b6c5eaf5` ↗.

4. System Design

This provides a description of the high-level components of the system and how they interact, including details like a function's externally controllable inputs and how an attacker could leverage each input to cause harm or which invariants or constraints of the system are critical and must always be upheld.

Not all components in the audit scope may have been modeled. The absence of a component in this section does not necessarily suggest that it is safe.

4.1. Component: MorphoPythOracleFactory

Description

The MorphoPythOracleFactory is a permissionless contract that stores the code for deploying the MorphoPythOracle contract and maintains the `isMorphoPythOracle` mapping. It implements the `createMorphoPythOracle` function, which allows any caller to deploy a new MorphoPythOracle instance.

The caller must provide the following arguments:

- `address pyth` — address of the Pyth contract on the current chain
- `IERC4626 baseVault` — address of the base vault, or zero address if not used
- `uint256 baseVaultConversionSample` — sample share amount for the base vault
- `bytes32 baseFeed1` — feed ID or `1`
- `bytes32 baseFeed2` — feed ID or `1`
- `uint256 baseTokenDecimals` — decimals for the base token
- `IERC4626 quoteVault` — address of the quote vault, or zero address if not used
- `uint256 quoteVaultConversionSample` — sample share amount for the quote vault
- `bytes32 quoteFeed1` — feed ID or `1`
- `bytes32 quoteFeed2` — feed ID or `1`
- `uint256 quoteTokenDecimals` — decimals for the quote token
- `uint256 priceFeedMaxAge` — max acceptable price age in seconds
- `bytes32 salt` — unique salt used for deterministic deployment

Test coverage

There are no test cases implemented for the MorphoPythOracleFactory contract.

Cases not covered

- A new MorphoPythOracle instance is successfully created via `createMorphoPythOracle`.
- The new oracle address is marked as `true` in the `isMorphoPythOracle` mapping.
- A `CreateMorphoPythOracle` event is emitted after successful deployment.
- Deployment with the same parameters (including `salt`) is not possible twice.

Attack surface

Since the `createMorphoPythOracle` function is permissionless and lacks input validation, any caller can specify arbitrary values for `pyth`, feeds, vaults, or other parameters, including malicious, incorrect, or invalid data.

4.2. Component: MorphoPythOracle

Description

The MorphoPythOracle is an oracle contract that uses the Pyth network to compute asset prices and implements the `price` function for this purpose.

The `constructor` performs validation on the provided parameters. Specifically, it checks that if the vault contract addresses are set to zero, the corresponding `ConversionSample` values must be equal to `1`. Additionally, both `baseVaultConversionSample` and `quoteVaultConversionSample` must be nonzero. The `SCALE_FACTOR` is precomputed using the provided `quoteTokenDecimals` and `baseTokenDecimals`, along with the feed decimals and the conversion-sample values for both base and quote vaults.

The `price` function returns the current `price` by combining the `SCALE_FACTOR`, data from Pyth oracle feeds, and the output of the `convertToAssets` function based on the specified vaults and conversion-sample amounts.

```
function price() external view returns (uint256) {
    return SCALE_FACTOR.mulDiv(
        BASE_VAULT.getAssets(BASE_VAULT_CONVERSION_SAMPLE)
            * PythFeedLib.getPrice(pyth, BASE_FEED_1, PRICE_FEED_MAX_AGE)
            * PythFeedLib.getPrice(pyth, BASE_FEED_2, PRICE_FEED_MAX_AGE),
        QUOTE_VAULT.getAssets(QUOTE_VAULT_CONVERSION_SAMPLE)
            * PythFeedLib.getPrice(pyth, QUOTE_FEED_1, PRICE_FEED_MAX_AGE)
            * PythFeedLib.getPrice(pyth, QUOTE_FEED_2, PRICE_FEED_MAX_AGE)
    );
}
```

If vaults or feeds are not specified (set to the zero address or a zero feed ID),

- for vaults, `getAssets(...)` returns `1`, and
- for feeds, `getPrice(...)` returns `1`.

This behavior allows the oracle to operate even when certain components (vaults or feeds) are intentionally omitted, offering flexibility in how the price is computed.

Invariants

- The `price()` function always returns a value ≥ 0.

- If `pyth.getPriceNoOlderThan` returns a negative price, the call reverts.
- The `pyth.getPriceNoOlderThan(priceId, PRICE_FEED_MAX_AGE)` guarantees the price data is no older than `PRICE_FEED_MAX_AGE`.
- Constructor parameters (`pyth_`, vault addresses, feed IDs, conversion samples, decimals, `PRICE_FEED_MAX_AGE`) are immutable and cannot be changed after deployment.

Test coverage

Cases covered

- Successful deployment of a MorphoPythOracle contract directly (without the MorphoPythOracleFactory).
- The `price` returns the expected value when using the `WBTC/USD` and `USDT/USD` feeds.

Cases not covered

- Negative-path tests.
- One vault omitted (base present, quote zero) but both feeds present.
- One feed omitted while both vaults are present.
- Both feeds omitted but vaults are present.
- The conversion samples are `1` and `type(uint256).max` to test rounding and `SCALE_FACTOR` calculation.
- Negative tests confirming that prices older than `PRICE_FEED_MAX_AGE` cause a revert.
- Verification that `SCALE_FACTOR` is computed correctly.

Attack surface

- Anyone can deploy a MorphoPythOracle through MorphoPythOracleFactory with arbitrary parameters. That means an attacker could supply a fake or malicious Pyth contract at deployment and gain full control over the `price`. Users therefore need to verify and fully trust the oracle instance they plan to use; see Finding 3.3. ↗.
- The `isMorphoPythOracle[oracle_addr] == true` only guarantees the bytecode is a MorphoPythOracle; it does not guarantee the configuration is safe. Even if a legitimate Pyth contract and valid feed IDs are provided, the price still depends on `vault.convertToAssets()`. Different vault designs, attacked vaults, or later changes to vault settings can affect the `convertToAssets` result and, as a result, the final price; see Finding 3.4. ↗.
- A further risk is the universal `PRICE_FEED_MAX_AGE`. One max age fits all feeds, which may be fine for some assets but too loose (or too strict) for others, leading either to stale prices or unnecessary reverts; see Finding 3.1. ↗.

5. Assessment Results

During our assessment on the scoped Pyth Morpho Wrapper contracts, we discovered seven findings. No critical issues were found. Two findings were of high impact, four were of medium impact, and the remaining finding was informational in nature.

We often observe a high number of findings in projects undergoing rapid development. The existing testing suite used during the assessment was limited in its coverage and did not fully test all functionalities of the smart contracts. This limitation prevented comprehensive testing and identification of potential issues, particularly in terms of negative testing scenarios. Some of the findings reported in this assessment could have been detected if the testing suite had been more comprehensive. Our recommendation to the Pyth Network team is to implement a security-focused development workflow. This includes augmenting the codebase with a comprehensive test suite to ensure proper behavior under real-world conditions.

To enhance the security and reliability of the protocol, we strongly recommend a thorough reaudit after development is complete and before deploying the protocol's contracts. This reaudit will help identify any potential issues or vulnerabilities, allowing for necessary changes or fixes prior to deployment.

5.1. Disclaimer

This assessment does not provide any warranties about finding all possible issues within its scope; in other words, the evaluation results do not guarantee the absence of any subsequent issues. Zellic, of course, also cannot make guarantees about any code added to the project after the version reviewed during our assessment. Furthermore, because a single assessment can never be considered comprehensive, we always recommend multiple independent assessments paired with a bug bounty program.

For each finding, Zellic provides a recommended solution. All code samples in these recommendations are intended to convey how an issue may be resolved (i.e., the idea), but they may not be tested or functional code. These recommendations are not exhaustive, and we encourage our partners to consider them as a starting point for further discussion. We are happy to provide additional guidance and advice as needed.

Finally, the contents of this assessment report are for informational purposes only; do not construe any information in this report as legal, tax, investment, or financial advice. Nothing contained in this report constitutes a solicitation or endorsement of a project by Zellic.