Auditing Report

Hardening Blockchain Security with Formal Methods

FOR

Lombard

Strategy Smart Contracts

Veridise Inc.
May 18, 2026

Last updated date: June 24, 2026

► **Prepared For:**

Lombard
https://www.lombard.finance/

► **Prepared By:**

Benjamin Mariano
Ajinkya Rajput

► **Contact Us:**

contact@veridise.com

► **Version History:**

June 24, 2026	V4
June 8, 2026	V3
June 2, 2026	V2
May 29, 2026	V1

Contents

About Veridise

Veridise is an independent security firm founded by academics and security researchers with deep expertise in formal methods, programming languages, and applied cryptography. The firm focuses on high-assurance security for blockchain and cryptographic systems, combining rigorous theory with practical adversarial analysis.

Veridise provides full-stack blockchain security services spanning smart contracts, zero-knowledge circuits and proving systems, consensus and execution clients, cryptographic libraries, and supporting infrastructure. The team routinely analyzes systems at the boundary between protocol design, implementation, and cryptography, where failures are both subtle and high-impact. To date, Veridise has conducted hundreds of security assessments for protocols securing billions of dollars in TVL.

Dynamic Security Veridise treats security as an ongoing engineering discipline rather than a one-time validation exercise. Effective security requires iterative review, continuous feedback, and close collaboration between auditors and system designers. Throughout the engagement, Veridise analysts communicated regularly with the Lombard team using AuditHub, enabling rapid clarification of design intent, prompt discussion of findings, and efficient validation of fixes.

In parallel with manual review, Veridise maintains active research programs focused on automated and formal analysis of blockchain systems. These efforts have produced tools such as *Vanguard* for static analysis and *Picus* for formal verification, which are available through AuditHub. AuditHub integrates these techniques directly into CI/CD workflows, supporting continuous vulnerability detection and regression analysis beyond the scope of a single audit.

Additional details and case studies are available at `https://audithub.dev/case-studies/`.

Veridise Reports Veridise reports are published in the public audit archive linked below. Only reports obtained from this archive or confirmed directly by a representative of Veridise should be considered official Veridise reports.

`https://veridise.com/audits-archive/`

1 Executive Summary

From May 18, 2026 to May 26, 2026, Lombard engaged Veridise to conduct a security assessment of their Strategy Smart Contracts. The security assessment covered the implementation of a strategy share system where users can deposit multiple supported assets through pricing converters, receive strategy shares, pay configurable deposit/redemption/management/performance fees, and redeem asynchronously through operator-fulfilled requests. Veridise conducted the assessment over 12 person-days, with 2 security analysts reviewing the project over 6 days on commits `3a34f90`, `da44075`, `5346539`, `43f31da`, `34b00df`, and `0b3d668` *. The review strategy involved a tool-assisted analysis of the program source code performed by Veridise security analysts as well as thorough code review.

Project Summary. The security assessment covered the smart contracts that implement the infrastructure for strategies, including the main strategy logic, blocklist, converters, shards, and the validator.

The main portion of the audit focuses on the strategy implementation itself, which implements an upgradeable ERC-20 share token representing ownership in a managed pool of underlying assets. Users can deposit any admin-approved asset, with each deposit asset priced through a configured converter. Deposits are optionally charged a fee before shares are minted. Redemptions from the strategy are asynchronous: users submit share redemption requests and an authorized redemption payer later fulfills requests by transferring underlying assets. The strategy also manages price-per-share updates, deposit/redeem/transfer pausing, transfer pausing, fee treasury configuration, management and performance fee harvesting, shard registration, and role-based administration.

The blocklist component provides compliance gating for strategy interactions. It combines a local blocklist, managed by authorized accounts, with external sanctions list contracts. Strategies can query this oracle to reject actions involving blocked or sanctioned accounts.

Converters normalize different supported deposit assets into the strategy's base accounting asset. There are a few different converters, including a "direct" converter that assumes a one-to-one value between deposit and base asset, and two different Chainlink-based converters that use Chainlink oracle feeds to perform the conversion.

Shards are auxiliary asset-holding contracts registered with the strategy. Shards can pull funds from the strategy, push funds back, and execute arbitrary calls when permitted. In general, shards provide the mechanism by which funds deposited in the strategy can be invested elsewhere, according to the limitations set out by the validator.

The validator component controls what shard execution calls are allowed. The validator is a generic implementation that allows the definition of "rulesets" that can place constraints on execution calls from shards, such as restrictions on receiver address, message value, function called, or even constraints on specific bytes in the calldata. The validator can dynamically update "rulesets", meaning restrictions can be updated as necessary for shards during execution.

* The code from `da44075` and `0b3d668` are limited to the changes introduced in this pull request and this pull request respectively (at the respective commits mentioned). The code from `5346539`, `43f31da`, and `34b00df` are limited to the code changes in this, this, and this respectively.

Code Assessment. The developers provided the source code of the Strategy Smart Contracts contracts for the code review. The source code appears to be mostly original code written by the Strategy Smart Contracts developers. It contains some documentation in the form of READMEs and documentation comments on functions and storage variables.

The source code contained a test suite, which the Veridise security analysts noted exercises both happy and adversarial paths. However, there are a number of adversarial paths which remain untested, such as unlisted asset behavior, bad manager/admin inputs, slippage limits exceeded, and already-fulfilled and non-existent redeem request fulfillment.

Summary of Issues Detected. The security assessment uncovered 20 issues, 1 of which is assessed to be of high severity by the Veridise analysts. Specifically, the issue V-LSTR-VUL-001 identifies that after migration of a vault the price per share is reset to a constant value that might be different from the price of shares in the source vault that is migrated. This may lead to loss or unfair gains for shareholders. The Veridise analysts also identified 3 medium-severity issues, including V-LSTR-VUL-002 and V-LSTR-VUL-003. V-LSTR-VUL-002 describes that enumeration of elements in an address set are assumed to be deterministic while they are not, while V-LSTR-VUL-003 details that there are no liveness checks on the values posted by the oracle used inside `StakedLBTCConverter`. Veridise analysts also identified 10 low-severity issues, 4 warnings, and 2 informational findings. The Strategy Smart Contracts developers have acknowledged all the issues and provided fixes for 14 issues and 1 issue are partially fixed and are being discussed.

Recommendations. After conducting the assessment of the protocol, the security analysts noticed that the strategy contracts expose a granular access-control model, with multiple roles responsible for configuration, operations, accounting, asset management, pausing, upgrades, and other privileged actions. This structure appears to rely on implicit assumptions about which parties will hold each role, such as Lombard-controlled addresses, strategy operators, custodians, or other third-party service providers. The protocol documentation should explicitly define which entity is expected to receive each role, what authority that role grants, and what trust assumptions users and integrators must make about that entity. In particular, any role that can affect deposits, withdrawals, pricing, asset conversion, strategy allocation, fee behavior, or emergency controls should be clearly documented as trusted, semi-trusted, or operationally-constrained.

Additionally, the number of privileged roles increases the protocol's operational key-management surface. Each privileged key represents a potential point of compromise or loss, and mistakes in role assignment could materially affect protocol safety. The protocol should require strict operational practices for all privileged operators, including the use of multisigs where appropriate, hardware-backed key management, least-privilege role assignment, clear separation of duties, documented key rotation and recovery procedures, and prompt revocation of unused or obsolete roles. These expectations should be part of the protocol's deployment and operations documentation rather than left as implicit assumptions.

Disclaimer. We hope that this report is informative but provide no warranty of any kind, explicit or implied. The contents of this report should not be construed as a complete guarantee that the system is secure in all dimensions. In no event shall Veridise or any of its employees be liable for any claim, damages or other liability, whether in an action of contract, tort or otherwise, arising from, out of or in connection with the results reported here.

2 Project Dashboard

Table 2.1: Application Summary.

Name	Version	Type	Platform
Strategy Smart Contracts	`3a34f90`	Solidity	Ethereum

Table 2.2: Engagement Summary.

Dates	Method	Consultants Engaged	Level of Effort
May 18–May 26, 2026	Manual & Tools	2	12 person-days

Table 2.3: Vulnerability Summary.

Name	Number	Acknowledged	Fixed
Critical-Severity Issues	0	0	0
High-Severity Issues	1	1	1
Medium-Severity Issues	3	3	2
Low-Severity Issues	10	10	7
Warning-Severity Issues	4	4	3
Informational-Severity Issues	2	2	1
TOTAL	20	20	14

Table 2.4: Category Breakdown.

Name	Number
Logic Error	14
Data Validation	3
Frontrunning	1
Flashloan	1
Maintainability	1

3 Security Assessment Goals and Scope

3.1 Security Assessment Goals

The engagement was scoped to provide a security assessment of Strategy Smart Contracts's source code. During the assessment, the security analysts aimed to answer questions such as:

- Can rounding allow attackers to game a strategy at the expense of other share holders?
- Can large shareholders manipulate a strategy to steal funds?
- Does the validator restrict shard executions as intended according to the rulesets provided?
- Are fees calculated correctly and fairly?
- Are trusted actions (e.g., price posting, rate setting, etc.) limited to trusted addresses?
- Are rate limits on price changes implemented appropriately to pause actions when price changes exceed expectations?
- Can a redemption request only be fulfilled once?
- Are sanctioned/blocklisted appropriately blocked from depositing and redeeming?
- Is the protocol vulnerable to common DeFi attack vectors, such as reentrancy, flashloan, etc.?

3.2 Security Assessment Methodology & Scope

Security Assessment Methodology. To address the questions above, the security assessment involved a combination of human experts and automated program analysis & testing tools. In particular, the security assessment was conducted with the aid of the following techniques:

- *Manual review.* Security analysts performed a line-by-line review of the in-scope code and supporting documentation to evaluate design assumptions and identify logic or access-control flaws.
- *Fuzzing/Property-based Testing*. Security analysts leveraged fuzz testing to determine if the protocol may deviate from the expected behavior. To do this, they formulated the desired behavior of the protocol [V] specifications. They then tested these [V] statements using Veridise's fuzzing framework OrCa, run through the AuditHub platform.

Scope. The scope of this security assessment is limited to the contracts in `contracts/strategy/`. It should be noted that in addition to auditing the implementations in that folder at `3a34f90`, Veridise auditors also reviewed the changes in this pull request which slightly adjusts the price-posting functionality to include adjustment based on shares added or removed since the price was recorded before calling the function and this pull request which makes a number of small changes and optimizations including adding checking of the blocklist on all calls to `_update`. Veridise auditors also reviewed the changes introduced by this commit, this commit, and this commit which introduced the ability to change the token name/symbol, added a getter for the deposit assets, and made some minor contract size optimizations respectively.

Methodology. Veridise security analysts inspected the provided tests and read the Strategy Smart Contracts documentation. They then began a review of the code assisted by both static analyzers and automated testing.

During the security assessment, the Veridise security analysts met with the Strategy Smart Contracts developers to ask questions about the code.

3.3 Classification of Vulnerabilities

When Veridise security analysts discover a possible security vulnerability, they must estimate its severity by weighing its potential impact against the likelihood that a problem will arise.

The severity of a vulnerability is evaluated according to the Table 3.1.

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

The likelihood of a vulnerability is evaluated according to the Table 3.2.

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

The impact of a vulnerability is evaluated according to the Table 3.3:

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 Trust Model

4.1 Operational Assumptions

In addition to assuming that any out-of-scope components behave correctly, Veridise analysts assumed the following properties held when modeling security for Strategy Smart Contracts:

- Rulesets used in the validator place reasonable restrictions on the executions of shards. What is "reasonable" may vary depending on the use case, but usually this should include things like limits on arbitrary token transfers and approvals and prohibition of calls to unknown/untrusted contracts.
- Rulesets used in the validator should be configured appropriately. In particular, we assume that ruleset creators implement rules that do not conflict with one another. Note that there are a few subtle ways that rulesets could be in conflict, including two different rules having conflicting constraints on a particular call or even a single rule having conflicting constraints on a particular set of bytes within the calldata. Ruleset creators should be particularly careful to note that the contract logic allows the user to specify a particular rule to validate a call against. This means that calls are not checked against *all* rules, but rather only the one specified by the caller. Thus, if there is one rule that would reject a call but another that would accept it, the caller can simply choose the one that would accept it. We assume that rulesets are made with these properties in mind.
- The price oracles used in the converters are trustworthy and manipulation resistant, as these implementations are out-of-scope for the current audit.
- The strategy maintainers are honest and not manipulatable. Maintainers do have some restrictions on their behaviors, such as rate limits on price changes, but ultimately a malicious strategy maintainer could still exploit their privileged actions to make profit at the expense of share holders.
- The function `totalAssets` computation rounds the value returned *down*. This function is not called within the reviewed code, but could be called by other contracts or off-chain components. We assume that rounding direction is accounted for in any such caller.
- Access control roles are appropriately set and maintained (see next section for more details).
- Blocklist checks are not desired on redeem fulfillment (it is checked only on redeem requests).
- If an account is blocklisted, its shares in the strategy are not recoverable by any actor, even a trusted owner, until the blocklist is changed to exclude the blocklisted account.
- The changes in `0b3d668` introduce checking of the blocklist on the sender and receiver of every `_update`. This means that the blocklist is enforced on transfers, which strengthens the notion of the blocklist to disallow blocklisted users from transferring their funds to a non-blocklisted account. However, it also can allow a rogue blocklist to brick critical operations such as deposits, redeems, and fee harvesting if critical address like the fee treasury are blocklisted. This is assumed to be an intended risk.

4.2 Privileged Roles

This protocol relies on a complex access control regime, in which a variety of privileged roles exist with different abilities. These roles include things like the `DEFAULT_ADMIN_ROLE`, which can grant other roles and take actions like registering shards, set the blocklist oracle address, and set various fees. These also include more limited roles, like the `PRIVILEGED_EXECUTOR_ROLE`, which is allowed to submit executions through the shard which are subject to the validator, the `PRICE_SETTER_ROLE` which can post price updates, and the `PAY_REDEMPTIONS_ROLE` which can fulfill submitted redemption requests. And there are many other roles we will not list explicitly here with a variety of limited but consequential abilities in the protocol.

In our review, we assume that each role is assigned to the desired and trusted entity. This means that the protocol could be at risk if roles are not properly assigned and/or if authorized roles are compromised. Protocol exploits arising from these roles acting outside of their privileged scope are considered out-of-scope.

Operational Recommendations. Highly-privileged, non-emergency operations should be operated by a multi-sig contract or decentralized governance system. These operations should be guarded by a timelock to ensure there is enough time for incident response. Highly-privileged, emergency operations should be tested in example scenarios to ensure the role operators are available and ready to respond when necessary.

Full validation of operational security practices is beyond the scope of this review. Users of the protocol should ensure they are confident that the operators of privileged keys are following best practices such as:

- Never storing a protocol key in plaintext, on a regularly used phone, laptop, or device, or relying on a custom solution for key management.
- Using separate keys for each separate function.
- Storing multi-sig keys in a diverse set of key management software/hardware services and geographic locations.
- Enabling 2FA for key management accounts. SMS should *not* be used for 2FA, nor should any account which uses SMS for 2FA. Authentication apps or hardware are preferred.
- Validating that no party has control over multiple multi-sig keys.
- Performing regularly scheduled key rotations for high-frequency operations.
- Securely storing physical, non-digital backups for critical keys.
- Actively monitoring for unexpected invocation of critical operations and/or deployed attack contracts.
- Regularly drilling responses to situations requiring emergency response such as pausing/unpausing.

5 Vulnerability Report

This section presents the vulnerabilities found during the security assessment. For each issue found, the type of the issue, its severity, location in the code base, and its current status (i.e., acknowledged, fixed, etc.) is specified. Table 5.1 summarizes the issues discovered:

Table 5.1: Summary of Discovered Vulnerabilities.

ID	Description	Severity	Status
V-LSTR-VUL-001	Incorrect migration price	High	Fixed
V-LSTR-VUL-002	‘getBlocklistEntries‘ incorrectly relies on . . .	Medium	Acknowledged
V-LSTR-VUL-003	The StakedLBTCConverter has no liveness . . .	Medium	Fixed
V-LSTR-VUL-004	Rounding in converter can lead to . . .	Medium	Fixed
V-LSTR-VUL-005	Potential frontrunning of price changes	Low	Acknowledged
V-LSTR-VUL-006	Reading past end of data in constraint . . .	Low	Fixed
V-LSTR-VUL-007	‘migrate‘ can be called twice	Low	Acknowledged
V-LSTR-VUL-008	Incorrect rounding direction on . . .	Low	Fixed
V-LSTR-VUL-009	Large stakeholder manipulation of . . .	Low	Fixed
V-LSTR-VUL-010	Management fee shares do not consider . . .	Low	Fixed
V-LSTR-VUL-011	Management fee accrual frequency . . .	Low	Fixed
V-LSTR-VUL-012	Strategy silently defaults to 18 decimals . . .	Low	Fixed
V-LSTR-VUL-013	NAV reconciliation treats fee-share mints . . .	Low	Acknowledged
V-LSTR-VUL-014	Reconciled PPS lets mid-epoch flows mask . . .	Low	Fixed
V-LSTR-VUL-015	Zero address is not blocklisted	Warning	Acknowledged
V-LSTR-VUL-016	Decimals underflow check could limit . . .	Warning	Fixed
V-LSTR-VUL-017	No check for valid deposit asset on ‘deposit‘ . . .	Warning	Fixed
V-LSTR-VUL-018	Possibly redemption blocking via valid . . .	Warning	Fixed
V-LSTR-VUL-019	Minor code suggestions	Info	Partially Fixed
V-LSTR-VUL-020	Merkle validator address fields use packed . . .	Info	Fixed

5.1 Detailed Description of Issues

5.1.1 V-LSTR-VUL-001: Incorrect migration price

Severity	High	**Commit**	N/A
Type	Logic Error	**Status**	Fixed
Location(s)	`contracts/strategy/MAWARStrategyMigrated.sol:71`		
Confirmed Fix At	smart-contracts/pull/453		

Description In `migrate`, when migrating from an existing vault, the price is set to `10 ** decimals()`. This disregards the current price of shares on the existing vault, which could result in a huge and unexpected change in price.

Impact This could cause a huge loss of funds for shareholders by incorrectly and unexpectedly increasing/decreasing the current share price.

Recommendation Add an argument to `migrate` for migrator-specified price value so that the trusted migrator can set the correct price.

Developer Response The developers added an argument to `migrate` for the price as suggested.

5.1.2 V-LSTR-VUL-002: 'getBlocklistEntries' incorrectly relies on non-deterministic element iteration

Severity	Medium	**Commit**	N/A
Type	Logic Error	**Status**	Acknowledged
Location(s)	`contracts/strategy/BlocklistOracle.sol:135`		
Confirmed Fix At	N/A		

Description The function `getBlocklistEntries` is intended to fetch up to `limit` entries in the `blocklistedAccounts` set, starting at index `offset`. The implicit assumption in this implementation is that indexing of elements in the set will remain the same over time, but this is not true. As a result, calls to this function may not return the expected blocklisted accounts.

The same issue appears in `getAllowlistEntries`.

Example Consider the following sequence and suppose the addresses `a0` to `a9` have been added to the blocklist and currently have indices `0` through `9` respectively.

1. `getBlocklistEntries(0, 5)` returns entries `[a0, a1, a2, a3, a4]`
2. `remove(a1)`
3. `getBlocklistEntries(5, 5)` will return entries `[a5, a6, a7, a8]` because `a9` will be moved to the index `1` in step 2

Impact It is unclear exactly how this function is used, but it is `external`, meaning other protocols or off-chain components may rely on this implementation. For those other protocols and components, this unexpected behavior could lead them to incorrectly check against the blocklist, possibly allowing blocklisted accounts to interact with those protocols/components when they should have been blocked.

Recommendation One option is to have off-chain components use the events to construct the current blocklist and remove this logic entirely. If this interface is required, implement proper on-chain pagination to maintain consistent indices for entries in the blocklist.

Developer Response The developers have acknowledged the issue but have decided not to fix at this time. They stated: "We accept that behaviour. The method only reads the storage. The caller should be aware that the array may be changed."

5.1.3 V-LSTR-VUL-003: The StakedLBTCConverter has no liveness checks

Severity	Medium	**Commit**	N/A
Type	Data Validation	**Status**	Fixed
Location(s)	`contracts/strategy/converters/StakedLBTCConverter.sol`		
Confirmed Fix At	smart-contracts/pull/477		

Description The protocol uses `StakedLBTCConverter` for LBTC prices. This converter uses an oracle to get the staking ratio. However, the `StakedLBTCConverter` does not check for time elapsed since the ratio was posted. Therefore, there are no liveness checks implemented in this converter.

Impact The `StakedLBTCConverter` can consume stale ratio values without having any way to know if the price is live or stale

Recommendation Only use an oracle that provides a mechanism to implement liveness checks and add relevant liveness checks to the converter.

Developer Response The developers have remove this converter and will use the ChainLink interface for this conversion instead.

5.1.4 V-LSTR-VUL-004: Rounding in converter can lead to significant lost funds

Severity	Medium	**Commit**	N/A
Type	Logic Error	**Status**	Fixed
Location(s)	`contracts/[...]/ChainlinkCompositeConverter.sol:103, 106, 118, 121`		
Confirmed Fix At	smart-contracts/pull/479		

Description The `ChainlinkCompositeConverter` is used to get the price of a token as number of token assets per base asset. The calculations in this contract accounts for differences in decimals of tokens and decimals of their Chainlink feeds as shown below.

```
function toBaseUnits(
    uint256 tokenAmount,
    Math.Rounding rounding
) external view override returns (uint256) {
    (uint256 num, uint256 den) = _readAnswers();
    if (!inverted) {
        uint256 cross = Math.mulDiv(tokenAmount, num, den, rounding);
        return Math.mulDiv(cross, scaleFwdNum, scaleFwdDen, rounding);
    } else {
        uint256 cross = Math.mulDiv(tokenAmount, den, num, rounding);
        return Math.mulDiv(cross, scaleInvNum, scaleInvDen, rounding);
    }
}
```

Snippet 5.1: Snippet from ChainlinkCompositConverter

The functions `toBaseUnit()` and `fromBaseUnit()` first calculate the intermediate value `cross` via either `Math.mulDiv(tokenAmount, num, den, rounding)` or `Math.mulDiv(tokenAmount, den, num, rounding)` depending on whether or not the feed is inverted. Because `tokenAmount` (the input token amount), `num` (the first feed amount), and `den` (the second feed amount) each can have separate decimals, it is possible for the rounding on these calls to substantially change values.

Example As an example, suppose the input token amount is 100,000 USD in USDC which has `6` decimals, so it is `1e11`. Suppose the first feed uses `6` decimals as well and indicates the price is 1-to-1, meaning the feed value is `1e6`. Now, suppose the final feed uses `18` decimals and also indicates the price is 1:1, meaning its value is `1e18`. The computation will be `Math.mulDiv(1e11, 1e6, 1e18, Math.rounding.Floor)` which will equal `0`. This would mean that despite the fact that the price should be 1:1, the user's 100,000 USD will be lost to rounding.

Impact As demonstrated above, with the right combination of decimals and values, a substantial amount of money can be lost due to rounding.

Recommendation The convertor contracts should normalize each feed to a predetermined decimal value before using them in computations. This will not remove losses due to rounding altogether, but should avoid cases like the example above where rounding results in substantial losses for users.

Developer Response The developers have refactored the code to reduce losses from rounding. There is still the potential of some rounding error resulting in losses, but the developers have decided this falls within acceptable bounds. There is also an additional risk that large decimal values and feed values could cause overflows for legitimate prices, but developers have indicated that they do not expect such values/prices to occur in practice.

5.1.5 V-LSTR-VUL-005: Potential frontrunning of price changes

Severity	Low	**Commit**	N/A
Type	Frontrunning	**Status**	Acknowledged
Location(s)	`contracts/strategy/StrategyBaseUpgradeable.sol:190`		
Confirmed Fix At	N/A		

Description The function `postPricePerShare` allows the trusted price setter to directly update the price-per-share which is used to determine how to convert assets into shares and vice-versa. Because the price update is not determined programmatically on every deposit/redeem, but instead is set by an outside operator via `postPricePerShare`, it is possible for users to frontrun price updates via `postPricePerShare` to either deposit/redeem depending on the direction of the price movement.

Impact The impact of frontrunning in this way is mitigated by a few factors. First, there is a cost to frontrunning, so the gains by a user would have to exceed the costs associated with frontrunning. Second, price updates are rate limited, meaning large changes in price are unlikely and will only happen at the explicit allowance of the Lombard protocol maintainers. Third, redemptions are asynchronous and are only fulfilled by a trusted entity, meaning malicious redemptions could be blocked from fulfillment. And finally, misbehaving users can be blocked via the `BlocklistOracle`, which should discourage bad behavior.

Given the above, there is a chance such an attack could occur if the right situation aligns (i.e., a user with large investment who has not be blocked is able to frontrun a relatively large price change). In that case, it could result in other shareholders eating the cost of the gains of the attacker, which could be large depending on the magnitude of price change and investment.

Recommendation There are a few mitigation techniques that could employed here, with varying degrees of effectiveness and difficulty of implementation.

One simpler approach would be to only allow price posting only during pauses of deposit/redeem. In that case, the protocol to update prices would be to pause, wait, update prices, and then unpause. The benefit of this approach would be that users could not deposit/redeem when they see the price change as operations would be paused. However, there could still be a rush after unpausing depending on the nature of the price change.

Another more complex approach would be a batch/epoch approach where deposit and redeem requests are handled during an epoch, settled against the same price-per-share, and are then posted. By adding a break between the deposit/redeem request and settlement against the price, users lose the ability to game this directly. However, this would be a somewhat significant code/logic change.

Developer Response The developers have acknowledged the issue but have chosen not to fix it. They stated: "We are aware of this risk. The mitigation will make PPS updates often and set up a reasonable deposit and redemption fee to prevent significant profit from abuse."

5.1.6 V-LSTR-VUL-006: Reading past end of data in constraint matching can lead to unexpected calldata passing

Severity	Low	**Commit**	N/A
Type	Logic Error	**Status**	Fixed
Location(s)	`contracts/[...]/MerkleAllowlistValidator.sol:211, 374`		
Confirmed Fix At	smart-contracts/pull/480		

Description In `_constraintsMatchCalldata`, the goal is to compare the function calldata against the given "constraints", which are really collections of bytes that indicate bits that must be set/unset in the calldata. The constraints work by indicating a specific word in the calldata (indicated by the "constraint offset" or `cOff`), indicating the bits of interest in that word (via the `mask`), and indicating the expected value of those bits of interest (via `expected`). This check is performed by the following logic:

```
if (
     (_wordAt(data, uint256(cOff) + _SELECTOR_LENGTH) & mask) !=
     expected
) {
     return false;
}
```

Snippet 5.2: Snippet showing implementation of constraints

This check works by fetching the word of the calldata `data` starting at the offset `cOff` and confirming that the bits indicated by `mask` are equal to the binary values indicated in `expected`.

The heart of the potential problem is in `_wordAt`, which simply fetches the word of the calldata starting at the specified offset. The interesting behavior occurs in the case that the offset is at a point where less than a word remains in the calldata. In that case, the function returns the final bytes followed by zero-padding for the rest of the 32 bytes.

This padding behavior can have a potentially unintended consequence when it comes to comparing to expected bytes, as the `expected` might expect `zero` values in certain location, which will be there *due to* padding even though they are not there.

Example As an example, suppose we have a constraint with offset `100`. Suppose `mask` is all ones (i.e., all bits are considered) and `expected` is a single one followed by all zeroes (i.e., the first bit is `1` and all others are `0`). All of the following cases will pass this constraint:

- `data` is length `105`, which means there is exactly one bit to be read at the word at offset `100` (because the actual offset is `100 + _SELECTOR_LENGTH` which is `104`). The last bit of `data` is `1`. In this case, `_wordAt` will read `1` as the first bit, then will right-pad zeroes to the end. This will end up matching `expected` exactly.
- `data` is length `106` with the last two bits being `10`.
- `data` is length `107` with the last three bits being `100`.
- etc.

Impact This could unintentionally allow a shorter calldata to pass a constraint in an unexpected way. Note that this should only occur if the calldata length constraint is not present, so variable calldata lengths are allowed by the rule.

Recommendation If this behavior is unintended, revert when reading beyond the end of the calldata. If this is intended, document it carefully as this could lead to subtle issues if not properly accounted for.

Developer Response The developers implemented a fix the will reject a rule if any constraint offset reads past the end of the calldata. Veridise analysts also suggest that the developers add an explicit revert in `_wordAt` if the call would read past the end of the calldata but this has not yet been implemented.

5.1.7 V-LSTR-VUL-007: ‘migrate‘ can be called twice

Severity	Low	**Commit**	N/A
Type	Logic Error	**Status**	Acknowledged
Location(s)	`contracts/strategy/MAWARStrategyMigrated.sol:46`		
Confirmed Fix At	N/A		

Description The function `migrate` is used to migrate the state for an existing vault to the new strategy-style protocol implemented in this code base. This function should only be called once on migration, but nothing prevents `DEFAULT_ADMIN_ROLE` from calling this multiple times.

Impact It seems like the intention might be that one upgrade after calling `migrate` to remove the `migrate` function preventing multiple calls. However, nothing programmatically enforces this is done. If this is done, important state could be entirely reset.

Recommendation Add a small amount of state that ensures `migrate` is only called once *or* clearly document the assumption that the contract should be updated after the migration to remove `migrate`.

Developer Response The developers have acknowledged this issue but have decided not to fix it. They stated: "We accept operational risk. The plan is never to keep this implementation and update it to regular in the same transaction."

5.1.8 V-LSTR-VUL-008: Incorrect rounding direction on 'convertToAssets'

Severity	Low	**Commit**	N/A
Type	Logic Error	**Status**	Fixed
Location(s)	`contracts/[...]/MultiAssetDepositUpgradeable.sol:None`		
Confirmed Fix At	smart-contracts/pull/473		

Description In `convertToAssets`, the function first calls `_convertToAssets` which converts the shares to base assets. Then, it calls `fromBaseUnits` to convert base assets to the specified deposit asset. When calling the first `_convertToAssets` to get base assets from shares, it correctly uses `Math.Rounding.Floor` which will *undercount* base assets if rounding is done. However, when converting the base assets to the deposit asset units, the call to `fromBaseUnits` uses `Math.Rounding.Ceil`. Typically, when computing the value of shares, one should round down in this case to avoid inflating the value of a share relative to assets – rounding up can allow a shareholder to slightly inflate their value at the expense of other shareholders.

Impact Because of the first round down on `_convertToAssets`, this will only result in an inflated share value if the amount gained from rounding up exceeds the amount lost from rounding down on the shares conversion. However, this can certainly happen. As a simple example, suppose the shares conversion has no rounding, but the conversion to deposit asset rounds up.

It also appears the protocol internally uses `_convertToAssets` and everywhere it recreates this logic it does round in the correct direction to protect the protocol. However, any external contract or off-chain component using this function could indicate the incorrect value.

Recommendation Change `Math.Rounding.Ceil` in call to `fromBaseUnits` to `Math.Rounding.Floor`.

Developer Response The developers implemented the suggested fix.

5.1.9 V-LSTR-VUL-009: Large stakeholder manipulation of performance/management fees

Severity	Low	**Commit**	N/A
Type	Flashloan	**Status**	Fixed
Location(s)	`contracts/strategy/AsyncRedemptionUpgradeable.sol:154`		
Confirmed Fix At	smart-contracts/pull/481		

Description For strategies, management and performance fees are computed proportional to the current total supply. This raises the concern of a large stakeholder being able to deposit a large amount, gather massive fees against the huge total supply, then immediately withdrawing their huge deposit.

Impact In order for this attack to be profitable, a huge stakeholder would either have to risk their funds being held up on the contract, or it would require a malicious user obtaining the `PAY_REDEMPTIONS_ROLE`. If the `PAY_REDEMPTIONS_ROLE` were compromised, it could also make this attack possible in a single transaction via a flashloan, making such a large stakeholder attack feasible beyond whales.

In the case that this attack happens, large amounts of shareholder value could be diluted by performance fees minted to the strategy maintainers.

Recommendation First, clarify the trust assumptions here. It is clear that strategy maintainers have a number of ways where they could earn an unfair profit at the expense of shareholders, such as malicious price updates. If they are also the `PAY_REDEMPTIONS_ROLE` and recipients of the performance fees, additional trust assumptions around this role should be explicitly enumerated. If `PAY_REDEMPTIONS_ROLE` is a separate entity with different trust assumptions, it would also be possible to only compute performance fees on price updates, which would reduce the ability of a malicious user to manipulate fees.

Developer Response The developers added a check that disallows a redeem request and fulfillment in the same block. As a result, flashloans cannot be used to obtain funds for the large stakeholder attack. It is still possible for a large stakeholder to perform this attack without a stakeholder, but they risk their funds being tied up in the contract if their behavior is flagged before fulfillment.

5.1.10 V-LSTR-VUL-010: Management fee shares do not consider mint dilution

Severity	Low	**Commit**	N/A
Type	Logic Error	**Status**	Fixed
Location(s)	`contracts/[...]/FeeManagerUpgradeable.sol:82-94, 186-199`		
Confirmed Fix At	smart-contracts/pull/487		

Description Management fee accrual computes fee shares against the pre-mint total supply, then mints exactly that amount to the management fee treasury. The configured annual management fee rate should correspond to the value share charged to strategy holders over the elapsed accrual period. A fee paid by minting shares must account for the fact that the newly minted fee shares dilute themselves as soon as they are created. The current formula mints `totalSupply * feeFraction`, so the treasury receives `feeFraction / (1 + feeFraction)` of the post-mint share supply instead of `feeFraction` of the pre-mint assets.

```
function _computeManagementFeeShares(
    uint48 lastFeeHarvestAt,
    uint16 bps,
    uint256 totalSupply
) internal view returns (uint256 shares, uint256 elapsed) {
    elapsed = block.timestamp - uint256(lastFeeHarvestAt);
    if (bps == 0 || totalSupply == 0 || elapsed == 0) return (0, elapsed);
    // bps <= 9_999 < 2^14, elapsed <= ~3.15e7 < 2^25 -> product < 2^39, no overflow
    shares = totalSupply.mulDiv(
        uint256(bps) * elapsed,
        _BPS_DENOMINATOR * _SECONDS_PER_YEAR,
        Math.Rounding.Floor
    );
}
```

Snippet 5.3: Snippet from `_computeManagementFeeShares` showing the accrued fee fraction applied directly to pre-mint supply

```
function _accrueManagementFee() internal virtual {
    FeeManagerStorage storage $ = _getFeeManagerStorage();
    (uint256 shares, uint256 elapsed) = _computeManagementFeeShares(
        $.lastFeeHarvestAt,
        $.managementFeeBps,
        totalSupply()
    );
    $.lastFeeHarvestAt = uint48(block.timestamp);
    if (shares == 0) {
        return;
    }
    _mint($.managementFeeTreasury, shares);
    emit ManagementFeeHarvested($.managementFeeTreasury, shares, elapsed);
}
```

Snippet 5.4: Snippet from `_accrueManagementFee` showing the computed shares minted without gross-up

For example, with 100 shares outstanding and a 10% management fee accrual, the function mints 10 fee shares. The treasury then owns `10 / 110 = 9.09%` of the strategy, not 10%. To mint

shares worth 10% of pre-mint NAV, the contract must mint `100 * 10% / (1 - 10%) = 11.11` shares.

This path is reached whenever management fee accrual is triggered before a user deposit, mint, redemption request, or fee-rate update. No privileged action is required for ordinary users to trigger the stale formula after time has elapsed.

Impact The management fee treasury receives less value than the configured annual management fee rate implies. This causes systematic protocol revenue undercollection and makes the effective fee depend on harvest timing.

Recommendation Compute management fee shares by solving for the target post-mint ownership fraction. For an accrued fee fraction `f`, mint `totalSupply * f / (1 - f)` shares rather than `totalSupply * f` shares.

Enforce this in `_computeManagementFeeShares`. The implementation should also handle accrual intervals where `bps * elapsed` approaches or exceeds the denominator, either by capping the interval, splitting the accrual into smaller periods, or explicitly rejecting unsupported fee fractions.

Developer Response The developers implemented the suggested change.

Veridise Note The fix subtracts `uint256(bps) * elapsed` from `_BPS_DENOMINATOR * _SECONDS_PER_YEAR`, which can revert if the former value becomes greater than the latter. The developers acknowledge this potential concern, but have decided it will not occur realistically as "even with a 10% fee it will be years without deposit/redeem operations to reach that situation".

5.1.11 V-LSTR-VUL-011: Management fee accrual frequency changes the effective fee rate

Severity	Low	**Commit**	N/A
Type	Logic Error	**Status**	Fixed
Location(s)	`contracts/strategy/FeeManagerUpgradeable.sol:187-247`		
Confirmed Fix At	smart-contracts/pull/474		

Description The protocol calculates fees as a percentage of `totalSupply`. So fee is charged for each share every time the fee accrual performed.

Additionally, the management fee BPS does not map to a single effective fee over a fixed time period because every accrual mints fee shares into `totalSupply`, and later accruals charge fees on those previously minted fee shares.

```
function _computeManagementFeeShares(
    uint48 lastFeeHarvestAt,
    uint16 bps,
    uint256 totalSupply
) internal view returns (uint256 shares, uint256 elapsed) {
    elapsed = block.timestamp - uint256(lastFeeHarvestAt);
    if (bps == 0 || totalSupply == 0 || elapsed == 0) return (0, elapsed);
    // bps <= 9_999 < 2^14, elapsed <= ~3.15e7 < 2^25 -> product < 2^39, no overflow
    shares = totalSupply.mulDiv(
        uint256(bps) * elapsed,
        _BPS_DENOMINATOR * _SECONDS_PER_YEAR,
        Math.Rounding.Floor
    );
}
```

Snippet 5.5: Snippet from `_computeManagementFeeShares` showing each accrual charging the elapsed fraction against the current `totalSupply`

```
function _accrueManagementFee() internal virtual {
    FeeManagerStorage storage $ = _getFeeManagerStorage();
    (uint256 shares, uint256 elapsed) = _computeManagementFeeShares(
        $.lastFeeHarvestAt,
        $.managementFeeBps,
        totalSupply()
    );
    $.lastFeeHarvestAt = uint48(block.timestamp);
    if (shares == 0) {
        return;
    }
    _mint($.managementFeeTreasury, shares);
    emit ManagementFeeHarvested($.managementFeeTreasury, shares, elapsed);
}
```

Snippet 5.6: Snippet from `_accrueManagementFee` showing the computed fee shares minted into supply before the next accrual

For example, with 100 shares and a 10% annual management fee, one annual accrual mints 10 shares. Two half-year accruals mint 5 shares first, then mint 5.25 shares on the new 105 share

supply, for 10.25 total shares. More frequent accrual approaches continuous compounding, so the total minted shares approaches `100 * (e^0.1 - 1) = 10.517...` before rounding.

Impact Shares of the strategy can be diluted by different amounts for the same advertised management fee BPS and elapsed time. More frequent accruals increase the fee shares minted to the management fee treasury, up to the compounding limit for the configured rate and period. This makes fee expectations and off-chain reporting non-deterministic and dependent on the fee accrual calls.

The impact is mainly economic clarity and disclosure. The behavior may be acceptable if the intended management fee is a compounding fee on current supply, but users and integrators need documentation that annual BPS is not frequency independent.

Recommendation Document the compounding semantics wherever management fee BPS is described. The documentation should state that fee shares minted by earlier accruals become part of `totalSupply` for later accruals, so more frequent accruals over the same period mint more total fee shares than a single accrual.

If the intended fee is frequency independent, change `_computeManagementFeeShares` or the fee-accounting model so the same BPS and elapsed time produce the same effective dilution regardless of accrual cadence.

Developer Response The developers clarified that this behavior is intended and added documentation clarifying this intended behavior.

5.1.12 V-LSTR-VUL-012: Strategy silently defaults to 18 decimals when asset decimals cannot be read

Severity	Low	**Commit**	N/A
Type	Data Validation	**Status**	Fixed
Location(s)	`contracts/strategy/StrategyBaseUpgradeable.sol:105-136`		
Confirmed Fix At	smart-contracts/pull/482		

Description `StrategyBaseUpgradeable` tries to get the decimals by making a low-level call to the `decimals()` function in the asset contract. If this call fails, i.e. the `success` value of this call is false, `StrategyBaseUpgradeable` sets the decimals of underlying asset to a default value of 18. This underlying may have decimals that are different than 18.

```
function __StrategyBase_init(
    string memory name_,
    string memory symbol_,
    uint48 initialDefaultAdminDelay_,
    address defaultAdmin_,
    IERC20 asset_,
    address blocklistOracle_
) internal onlyInitializing {
  ...

    StrategyBaseUpgradeableStorage
        storage $ = _getStrategyBaseUpgradeableStorage();
    (bool success, uint8 assetDecimals) = _tryGetAssetDecimals(asset_);
    $.underlyingDecimals = success ? assetDecimals : 18;
  ...
}
```

Snippet 5.7: Snippet from `__StrategyBase_init` showing the silent 18-decimal fallback

Impact A strategy can be initialized with an incorrect share precision and PPS scale without any visible failure. Operators, price setters, wallets, and external protocols may then scale share amounts or posted PPS values using different assumptions. This can lead to materially incorrect deposits, redemptions, price reporting, or collateral valuation when the underlying asset is not actually 18 decimals.

Recommendation Remove the silent fallback. Initialization should revert when the underlying asset's decimals cannot be read, or it should require an explicit `assetDecimals_` initializer argument.

Developer Response The developers added an argument for the decimals that is used in the event fetching the decimals from the contract fails.

5.1.13 V-LSTR-VUL-013: NAV reconciliation treats fee-share mints as asset-backed deposits

Severity	Low	**Commit**	N/A
Type	Logic Error	**Status**	Acknowledged
Location(s)	`contracts/strategy/StrategyBaseUpgradeable.sol:123`		
Confirmed Fix At	N/A		

Description The `NAV` implementation of `postPricePerShare` treats every supply increase after the operator snapshot as an asset-backed deposit, but fee modules can mint shares without adding assets. User deposit shares are backed by the deposited net assets. The reconciliation issue is that those backed deposit shares and the unbacked fee shares minted in the same window are collapsed into one `currentSupply - supplyAtSnapshot` delta. The branch computes `sharesAdded = currentSupply - supplyAtSnapshot` and adds `sharesAdded * oldPPS` to `navAtSnapshot`, so management or performance fee shares minted between the snapshot and the NAV post create NAV without backed assets.

```
            if (currentSupply >= supplyAtSnapshot) {
                uint256 sharesAdded = currentSupply - supplyAtSnapshot;
                reconciledNav =
                    navAtSnapshot +
                    sharesAdded.mulDiv(
                        $.pricePerShare,
                        oneShare,
                        Math.Rounding.Floor
                    );
            } else {
                uint256 sharesRemoved = supplyAtSnapshot - currentSupply;
                reconciledNav =
                    navAtSnapshot -
                    sharesRemoved.mulDiv(
                        $.pricePerShare,
                        oneShare,
                        Math.Rounding.Ceil
                    );
            }
```

Snippet 5.8: Snippet from `postPricePerShare` showing supply additions being added to snapshot NAV at the old PPS

For example, if the snapshot records 100 assets backing 100 shares at PPS 1, and 10 fee shares are minted before `postPricePerShare`, the correct active PPS is `100 / 110 = 0.909...`. The branch computes `reconciledNav = 100 + 10 * 1 = 110` and stores PPS 1, so `totalAssets()` reports 110 assets even though no new assets entered the strategy.

Impact External lending markets, vaults, and risk engines that rely on `pricePerShare()` or `totalAssets()` can overvalue strategy shares after fee accrual. The overvaluation can let borrowers extract excess credit or let integrations accept undercollateralized positions until a later correction.

Recommendation The NAV reconciliation should account for the type of supply delta, not only the raw difference between current and snapshot supply. Fee-share mints should dilute PPS or be excluded from the asset-backed `sharesAdded` adjustment.

Developer Response The developers have acknowledged this issue. They stated: "We accept that due to shares minted as fees, total assets can be overvalued. The operational mitigations are applying reasonable fee bps, which will be covered by accrued rewards. As a result, the overvaluation will be corrected by the next PPS update."

5.1.14 V-LSTR-VUL-014: Reconciled PPS lets mid-epoch flows mask rate-limit violations

Severity	Low	**Commit**	N/A
Type	Logic Error	**Status**	Fixed
Location(s)	`contracts/strategy/StrategyBaseUpgradeable.sol:123`		
Confirmed Fix At	smart-contracts/pull/488		

Description The implementation of `postPricePerShare` with NAV posting applies the PPS rate limiter to the *reconciled* `priceToSet`, even though the documentation says the limiter should gate the declared snapshot PPS move *before* mid-epoch moves are considered. If the goal is to rate limit before reconciliation with more recent deposits/redeems, the rate limiter should measure the operator-declared NAV move from `oldPPS` to `navAtSnapshot * oneShare / supplyAtSnapshot`. Instead, deposits or redemptions that occur after the snapshot can change `currentSupply`, reduce the final `priceToSet` movement, and let a large declared NAV shock pass the limiter.

```
        uint256 changeBps = PriceRateLimiter._relativeChangeBps(
            $.pricePerShare,
            priceToSet
        );
        if (!$.ppsRateLimiter._wouldExceedLimit(changeBps)) {
            $.ppsRateLimiter._consume(changeBps);
            _setPricePerShare($, priceToSet);
            return;
        }
```

Snippet 5.9: Snippet from `postPricePerShare` showing the rate limiter consuming change against `priceToSet`

For example, assume old PPS is 1, the snapshot observes 200 assets backing 100 shares, and the limiter should reject a 100% declared increase. If 100 new shares are minted at stale PPS before posting, the branch reconciles to `(200 + 100) / 200 = 1.5`. A limiter configured below 100% but above 50% allows the post, even though the off-chain NAV snapshot reported a 100% move for the pre-existing shares.

Impact The circuit breaker can fail to pause or reject NAV updates that external protocols expect it to catch. An attacker or opportunistic depositor can use same-epoch supply changes to reduce the visible PPS movement and bypass limits around large gains or losses. Integrations that treat the rate limiter as protection against sudden oracle/NAV shocks can accept a price update that should have exhausted the bucket or paused strategy operations.

Recommendation If the desire is to catch large movements before reconciliation, compute the rate-limit delta against the declared snapshot PPS before applying same-epoch reconciliation. The reconciled PPS can still be stored, but `changeBps` should use `declaredPps = navAtSnapshot * oneShare / supplyAtSnapshot`. Enforce this in `postPricePerShare` and keep the documentation aligned with the implemented behavior. Add tests where large declared NAV increases and decreases are masked by deposits or redemptions, and assert that the rate limiter still rejects or pauses based on the declared move.

Otherwise, if the desire is to include mid-epoch deposits/redeems in rate limiting, adjust documentation to appropriately reflect this behavior.

Developer Response The developers updated the logic to detect price variations before reconciliation, as indicated by the comments.

Veridise Note It should be noted that this change does mean that deposits/redeems could cause a more extreme price shift that does not trigger rate limiting.

5.1.15 V-LSTR-VUL-015: Zero address is not blocklisted

Severity	Warning	**Commit**	N/A
Type	Logic Error	**Status**	Acknowledged
Location(s)	`contracts/strategy/BlocklistOracle.sol:125`		
Confirmed Fix At	N/A		

Description The function `isBlocklisted` will always return `false` for the zero address. The function `blockAccount` does not allow adding the zero address, so there is no way to add it to the `blocklistedAccounts` set.

Impact It is not possible to blocklist the zero address.

Recommendation Make it possible to blocklist the zero address as this may be desired in some cases.

Developer Response The developers acknowledged the issue but decided they will not have a use case where blocklisting the zero address is necessary.

5.1.16 V-LSTR-VUL-016: Decimals underflow check could limit supportable tokens

Severity	Warning	**Commit**	N/A
Type	Logic Error	**Status**	Fixed
Location(s)	`contracts/strategy/[...]/ChainlinkConverter.sol:90-100`		
Confirmed Fix At	smart-contracts/pull/486		

Description In the constructor of the `ChainlinkConverter`, the `scalingFactor` is computed either as `uint256(feedDec) + baseDecimals_ - tokenDecimals` or `uint256(feedDec) + tokenDecimals_ - baseDecimals_` depending on whether or not the converter is "inverted".

These checks can revert if the subtraction cause an overflow, which the logic explicitly checks for and reverts with `ChainlinkConverter_DecimalsUnderflow()` in that case.

This restriction might make this converter unable to handle certain token pairs. For example, suppose the deposit token is ETH with 18 decimals, the base token is BTC with 8 decimals, and we are using a BTC/ETH feed with 8 decimals. This will lead to: 8 + 8 - 18 which fails the check.

Impact In some cases, it may be possible to avoid this issue by using the `inverted_` flag to invert the calculation, which will avoid the underflow. However, this is only possible if the corresponding inverted feed is available. If no such inverted feed is available, there may be some token/feed combinations that cannot be supported.

Recommendation There are a few different mitigations.

One is to add the subtracted decimals to the denominator in this case, instead of reverting on the underflow. Note that this approach may change the nature of rounding and care should be taken to ensure that rounding is done appropriately.

Another is to simply document this limitation and provide documentation explaining the approach for dealing with such combinations (perhaps such as using the `ChainlinkCompositeConverter`).

Developer Response The fix adds supports for the potentially unsupported tokens.

5.1.17 V-LSTR-VUL-017: No check for valid deposit asset on 'deposit' paths

Severity	Warning	**Commit**	N/A
Type	Data Validation	**Status**	Fixed
Location(s)	`contracts/strategy/MultiAssetDepositUpgradeable.sol:223`		
Confirmed Fix At	smart-contracts/pull/483		

Description Strategies allow users to deposit in multiple tokens. Each token that is supported is explicitly added by the strategy maintainers and deposits should only be supported for these added tokens. For calls to `deposit`, it is possible to have a successful call with a deposit asset that has not been added by the strategy maintainers, as there is no explicit check that the deposit asset is supported.

Impact Instead of an explicit check that the deposit asset is supported, the logic instead uses `previewDeposit` on calls to `deposit` and `previewMint` on calls to `mint` to determine the amount of shares/assets the provided amount corresponds to. For both functions, if the deposit asset is unsupported, the preview functions will return `0`. For `mint`, this will result in `0` assets which will fail at the check `if (netAssets == 0) revert Strategy_ZeroAmount();` in `_deposit`. However, for `deposit`, this will result in non-zero assets but `0` shares. In that case, the call to `_deposit` will succeed. Because shares is `0`, it doesn't allow the attacker to mint any shares. However, they can cause the protocol to emit unexpected events, which could disturb off-chain components relying on these events.

Recommendation Add an explicit check disallowing mints/deposits using unsupported assets.

Developer Response The developers added a check that the amount of shares for a deposit is non-zero. This prohibits the use of unsupported token indirectly, as using one on a `deposit` path will set the shares to zero using the current logic.

5.1.18 V-LSTR-VUL-018: Possibly redemption blocking via valid redemptions

Severity	Warning	**Commit**	N/A
Type	Logic Error	**Status**	Fixed
Location(s)	`contracts/strategy/AsyncRedemptionUpgradeable.sol:168`		
Confirmed Fix At	smart-contracts/pull/455		

Description In `fulfillRedeemRequests`, the function will revert if any of the request IDs included has an asset amount of `0`. Because of rounding, it is possible that a legitimate attempt to redeem could result in `0` assets. This would mean trying to fulfill a legitimate request would cause this function to revert.

Impact This could block redemption requests, especially if the process of fulfilling requests is performed by some off-chain component which monitors events from calls to `requestRedeem`.

This could also lead to small amounts of shares/assets that cannot be removed from totals as expected.

Recommendation Make sure that legitimate request IDs with asset amounts `0` are filtered from calls to `fulfillRedeemRequests` (acknowledging the totals will never be decremented) or allow fulfillment of 0-asset requests.

Alternatively, make sure redeem requests with amount `0` are reverted at the request stage rather than on fulfillment.

Developer Response The developers added the suggested fix to reject redeem requests with zero assets.

5.1.19 V-LSTR-VUL-019: Minor code suggestions

Severity	Info	**Commit**	N/A
Type	Maintainability	**Status**	Partially Fixed
Location(s)	`contracts/strategy/` ▶ `BlocklistOracle.sol:104` ▶ `MAWARStrategyMigrated.sol:21, 67` ▶ `StrategyBaseUpgradeable.sol:288, 439, 468` ▶ `converters/` • `ChainlinkConverter.sol:130` • `StakedLBTCConverter.sol:48` ▶ `libraries/PriceRateLimiter.sol:29`		
Confirmed Fix At	smart-contracts/pull/484		

Description The auditors identified the following minor code improvements:

1. In `removeSanctionList` in the `BlocklistOracle`, there is no non-zero check for the sanction list being removed. Technically this is not necessary at the moment because it will revert on the `remove` as zero addresses cannot be added to the `sanctions` set, but it may be advisable to add an explicit check in this case for a more clear error report.
2. In `_getAnswer` in the `ChainlinkConverter`, there is no check that the round ID is non-zero. This is an extra sanity check that can be added to ensure that the answer returned by the feed is valid.
3. In `fromBaseUnits` in `StakedLBTCConverter`, there is no check that the ratio is non-zero (unlike `toBaseUnits`).
4. In `__StrategyBase_init`, `_tryGetAssetDecimals` is used on the provided asset to get the underlying decimals, while in `migrate` in `MAWARStrategyMigrated`, a separate `decimals_` argument is provided. These should share the same approach.
5. The constant `ENTERED` in `MAWARStrategyMigrated` is unused.
6. In `registerShard` in `StrategyBaseUpgradeable`, there is no check that the shard being registered is non-zero.
7. In `StrategyBaseUpgradeable`, the function `_pauseDepositAndRedeemForRateLimit` is unused.
8. In `PriceRateLimiter`, the constant `BPS_DENOMINATOR` is different than the constant `_BPS_DENOMINATOR` from `StrategyBaseUpgradeable`. These names are very close and could be easily confused. It would make sense to differentiate these names, like calling the one in `PriceRateLimiter` something like `MICRO_BPS_DENOMINATOR`.
9. The `TRANSFERS_PAUSER_ROLE` can do more than just pause transfers. They can call the general `pause` function which can also pause price posting. It might make sense to rename this role to indicate this impact it can have.
10. `changeNameAndSymbol` allows the default admin to change the name and symbol for the ERC20 token. While this is not directly buggy, it may be unexpected by certain frontends, marketplaces, etc. As a result, we suggest carefully documenting this behavior and ensuring that only trusted actors can trigger this only when necessary.

Impact Minor code issues including incorrect comments, confusing code, or incomplete checks can compound over time in a codebase. While these issues are now deemed to be minor in impact, they may cause issues in the future that could lead to serious vulnerabilities.

Recommendation Make the suggested minor improvements.

Developer Response The developers implemented some of the suggested fixes, skipping (3), (4), and (9). (7) was addressed in another commit according to developers.

5.1.20 V-LSTR-VUL-020: Merkle validator address fields use packed encoding instead of ABI address encoding

Severity	Info	**Commit**	N/A
Type	Logic Error	**Status**	Fixed
Location(s)	`contracts/strategy/[...]/MerkleAllowlistValidator.sol:398-405`		
Confirmed Fix At	smart-contracts/pull/485		

Description `MerkleAllowlistValidator` reads rule header addresses from the first 20 bytes of a 32-byte calldata word, so standard ABI address encoding cannot be used for `sender` and `to` fields in `ruleEncoded`. Rule builders should encode these fields as raw 20-byte packed addresses at `_RULE_OFFSET_SENDER` and `_RULE_OFFSET_TO`. ABI encoding stores an address as `[12 zero bytes][address: 20 bytes]`, but the validator's custom rule format expects `[address: 20 bytes][trailing field bytes]` at the configured field offset.

```
function _readAddress(
    bytes calldata r,
    uint256 s
) private pure returns (address a) {
    assembly {
        a := shr(96, calldataload(add(r.offset, s)))
    }
}
```

Snippet 5.10: Snippet from `_readAddress` showing the packed-address read

Impact Merkle tree generators, deployment scripts, rule-authoring UIs, and other components can generate invalid Merkle leaves if they treat the rule header as standard ABI-encoded data. Those leaves will either fail canonical checks, fail to match the intended sender or target, or encode a different truncated-looking address value than the tool author expected that passes the check.

Recommendation Document the rule header as a packed binary format and explicitly state that `sender` and `to` must be encoded as raw 20-byte addresses. The documentation should include the contrast with standard ABI encoding:

```
Standard ABI address slot: [12 zero bytes][address: 20 bytes]
Validator rule address:   [address: 20 bytes][next field bytes]
```

Developer Response Developers have added documentation for this encoding.