ARCHITECTURE NOTICE — READ BEFORE PROCEEDING

This audit (dated 28 February 2026) examined Minted's prior cross-chain DAML + Solidity architecture, which included an Ethereum bridge (BLEBridgeV9, Wormhole integration). **That architecture has been fully deprecated.**

Minted's current production architecture is 100% Canton-native with no Ethereum bridge, no Wormhole relayer, and no cross-chain components.

Bridge-related findings in this report (sections 6.2.5, and any references to BLEBridgeV9 or Wormhole) are **NOT APPLICABLE** to the current architecture.

Audit Results Summary

• **No critical issues found**

• 4 high severity issues — **all FIXED**

• 9 medium severity issues — **all FIXED or ACKNOWLEDGED**

• 12 low severity issues — **all FIXED**

• Executive summary: "The Minted team has successfully addressed all identified issues from the audit. All vulnerabilities have been mitigated."

Current Architecture Documentation

Canonical deck: https://deck.minted.app/

Technical architecture: https://deck.minted.app/minted-technical-architecture.pdf

softstack

Smart Contract Code Review And Security Analysis Report

CUSTOMER	Minted
SCOPE	Core Contracts / DeFi Lending
DATE	28.02.2026

Made in Germany by softstack GmbH

Table of Contents

1. Disclaimer

The audit makes no statements or warrantees about utility of the code, safety of the code, suitability of the business model, investment advice, endorsement of the platform or its products, regulatory regime for the business model, or any other statements about fitness of the contracts to purpose, or their bug free status. The audit documentation is for discussion purposes only.

The information presented in this report is confidential and privileged. If you are reading this report, you agree to keep it confidential, not to copy, disclose or disseminate without the agreement of the client. If you are not the intended receptor of this document, remember that any disclosure, copying or dissemination of it is forbidden.

Version / Date	Description
0.1 (15.02.2026)	Layout
0.5 (20.02.2026)	Manual + Automated Security Testing
1.0 (28.02.2026)	Final document
1.1 (28.02.2026)	Re-check

2. About the Project and Company

Company: MintedAssociates Corp

Address: 511 South DuPont Highway Ste 10, Dover, Delaware 19901, USA

Website: https://www.minted.app/

Twitter (X): https://x.com/tryMinted

LinkedIn: https://www.linkedin.com/company/mintedmarketplace

Instagram: https://www.instagram.com/tryminted/

Minted is a decentralized private equity platform that provides the first unified financial instrument bridging the gap between open decentralized market trading and legal shareholder rights. The platform enables real equity to be tokenized, raised, traded, and made liquid through a globally compliant architecture.

At the core of Minted's technology is an on-chain **equity state switch** embedded into every asset, allowing holders to move seamlessly between frictionless speculation and full shareholder rights without intermediaries. This proprietary infrastructure unifies compliant fundraising, equity access, DEX trading, and cap-table synchronization into a single system deployable on any permissionless blockchain.

Key capabilities include:

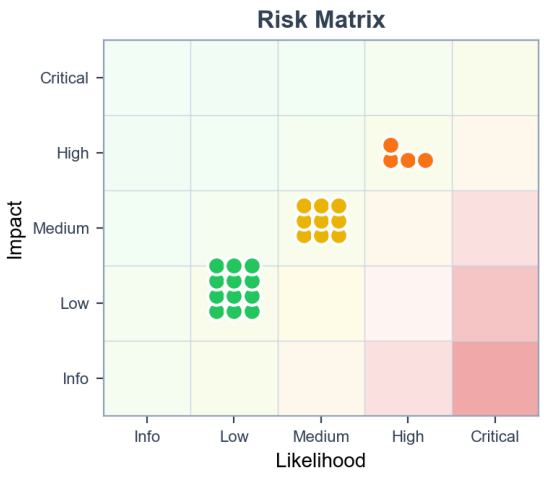
- **Equity Tokenization** — Founders create equity-backed tokens tied directly to Net Asset Value (NAV) through US Nexus or SPV structures.
- **Compliant Offerings** — Fundraising and token issuance are executed through licensed compliance partners, including a US Special Purpose Broker-Dealer (SPBD) with real-time reporting.
- **State Transitions** — Investors can instantly switch between permissionless trading and exercising full shareholder rights on-chain.
- **Automated Cap-Table Sync** — Post-raise equity beneficiary management synchronizes automatically via Minted's "Equity Locked Environment" with institutional-grade reporting.

The smart contracts under audit form the core DeFi lending infrastructure of the Minted platform, handling deposit routing, collateral vaults, borrowing modules, liquidation engines, treasury management, and yield strategy integrations.

3. Vulnerability & Risk Level

Risk represents the probability that a certain source-threat will exploit vulnerability, and the impact of that event on the organization or system. Risk Level is computed based on CVSS version 3.1.

Level	CVSS Score	Vulnerability	Required Action
CRITICAL	9.0 – 10.0	A vulnerability that can disrupt the contract functioning in a number of scenarios, or creates a risk that the contract may be broken.	Immediate action to reduce risk level.
HIGH	7.0 – 8.9	A vulnerability that affects the desired outcome when using a contract, or provides the opportunity to use a contract in an unintended way.	Implementation of corrective actions as soon as possible.
MEDIUM	4.0 – 6.9	A vulnerability that could affect the desired outcome of executing the contract in a specific scenario.	Implementation of corrective actions in a certain period.
LOW	0.1 – 3.9	A vulnerability that does not have a significant impact on possible scenarios for the use of the contract and is probably subjective.	Implementation of certain corrective actions or accepting the risk.
INFO	0.0	A vulnerability that has informational character but is not affecting any of the code.	An observation that does not determine a level of risk.



4. Auditing Strategy and Techniques Applied

Throughout the review process, care was taken to evaluate the repository for security-related issues, code quality, and adherence to specification and best practices.

The auditing process follows a routine series of steps:

Code review that includes the following:

- Review of the specifications, sources, and instructions provided to softstack to make sure we understand the size, scope, and functionality of the smart contract.
- Manual review of code, which is the process of reading source code line-by-line in an attempt to identify potential vulnerabilities.
- Comparison to specification, which is the process of checking whether the code does what the specifications, sources, and instructions provided to softstack describe.

Testing and automated analysis that includes the following:

- Test coverage analysis, which is the process of determining whether the test cases are actually covering the code and how much code is exercised when we run those test cases.
- Symbolic execution, which is analysing a program to determine what inputs causes each part of a program to execute.

Best practices review, which is a review of the smart contracts to improve efficiency, effectiveness, clarify, maintainability, security, and control based on the established industry and academic practices, recommendations, and research.

Specific, itemized, actionable recommendations to help you take steps to secure your smart contracts.

5. Metrics

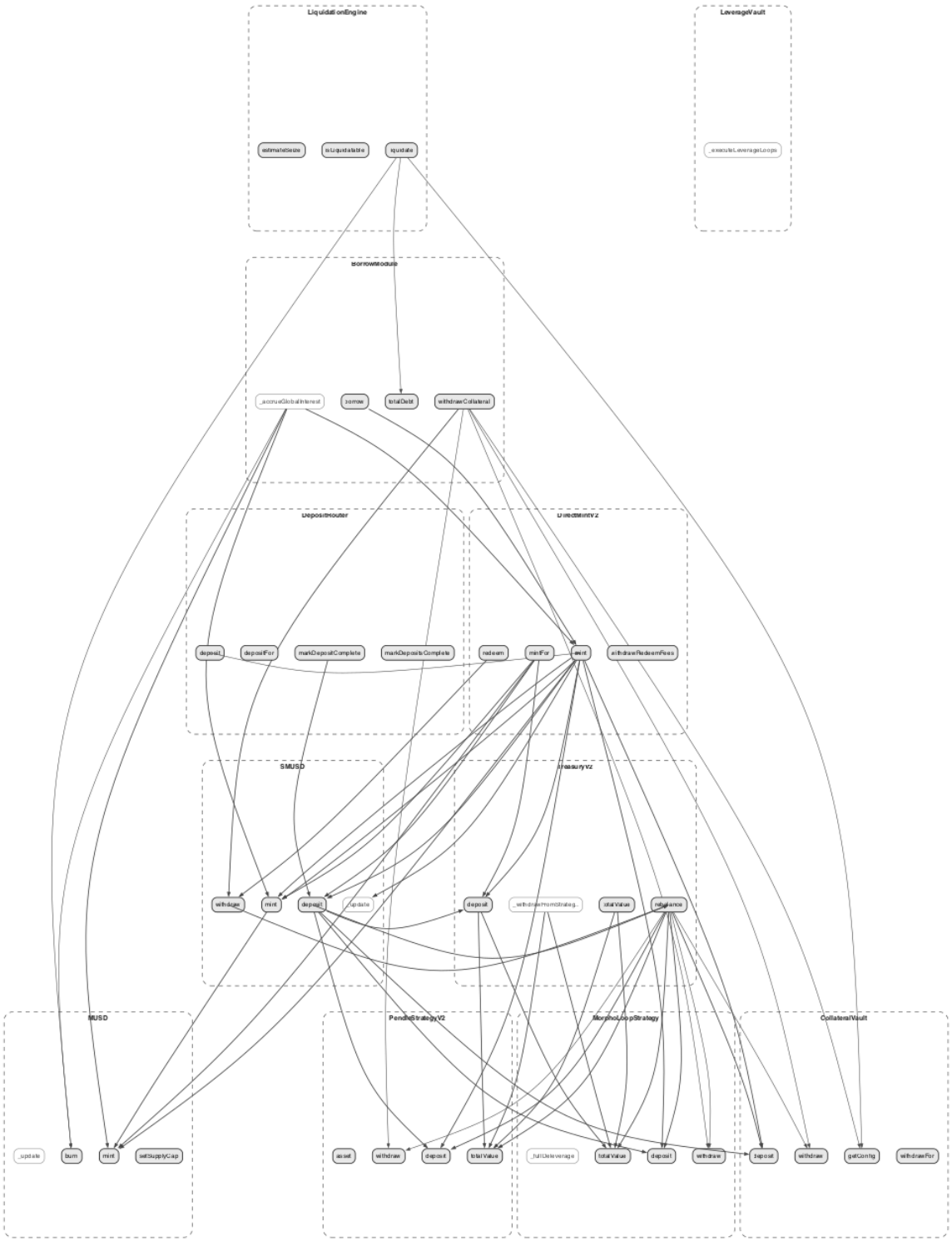
The metrics section gives the reader an overview of the size, quality, flows and capabilities of the codebase.

5.1 Tested Contract Files

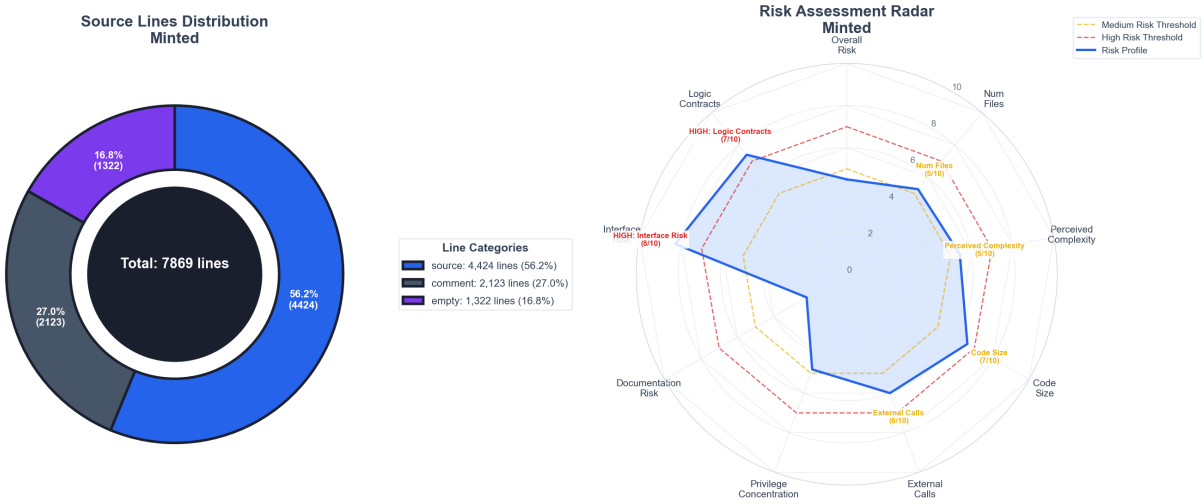
Source: https://github.com/luthatdude/Minted-mUSD-Canton
Commit: `393fd96f322674b25dd4b0c21f593f0874b86211`

File	Fingerprint (MD5)
./contracts/BLEBridgeV9.sol	`3a98c72c5ccf101dffd193ea46936f0b`
./contracts/BorrowModule.sol	`4c019e3d649f59d9732a845b4334953c`
./contracts/CollateralVault.sol	`b37713d70d939e21a31b561431f140ba`
./contracts/DepositRouter.sol	`61a3f69287a6f34f7deff412a1091d9e`
./contracts/DirectMintV2.sol	`bad145b58c6882f498338200295c853c`
./contracts/InterestRateModel.sol	`b3e790eb47ac9f86d61019489faac0d1`
./contracts/LeverageVault.sol	`288f70aaf405d17d611f7a2d547b4d60`
./contracts/LiquidationEngine.sol	`fceee70cdc68ae2e2a21fc99ca391df5`
./contracts/MUSD.sol	`52c873366b875015c37c181617f16af7`
./contracts/PendleMarketSelector.sol	`1c5e3f00cf19959fbf89ee664e378782`
./contracts/PriceOracle.sol	`5298ef79f700a43bf3d46a39391069a1`
./contracts/SMUSD.sol	`e184e0973f18d0e82b8568938c85fd7b`
./contracts/TreasuryReceiver.sol	`509f619d8ef778062e891171bfedcfbc`
./contracts/TreasuryV2.sol	`51df78fa8784d2fbbbc57364af684aca`
./contracts/interfaces/IStrategy.sol	`60e95c37a43d6cc32764bb5d4eed67af`
./contracts/strategies/MorphoLoopStrategy.sol	`f6eff6350335c2e38a2c112067328978`
./contracts/strategies/PendleStrategyV2.sol	`efc6cd8bb70701e6599dc81efbdaa519`

5.2 Call Graph



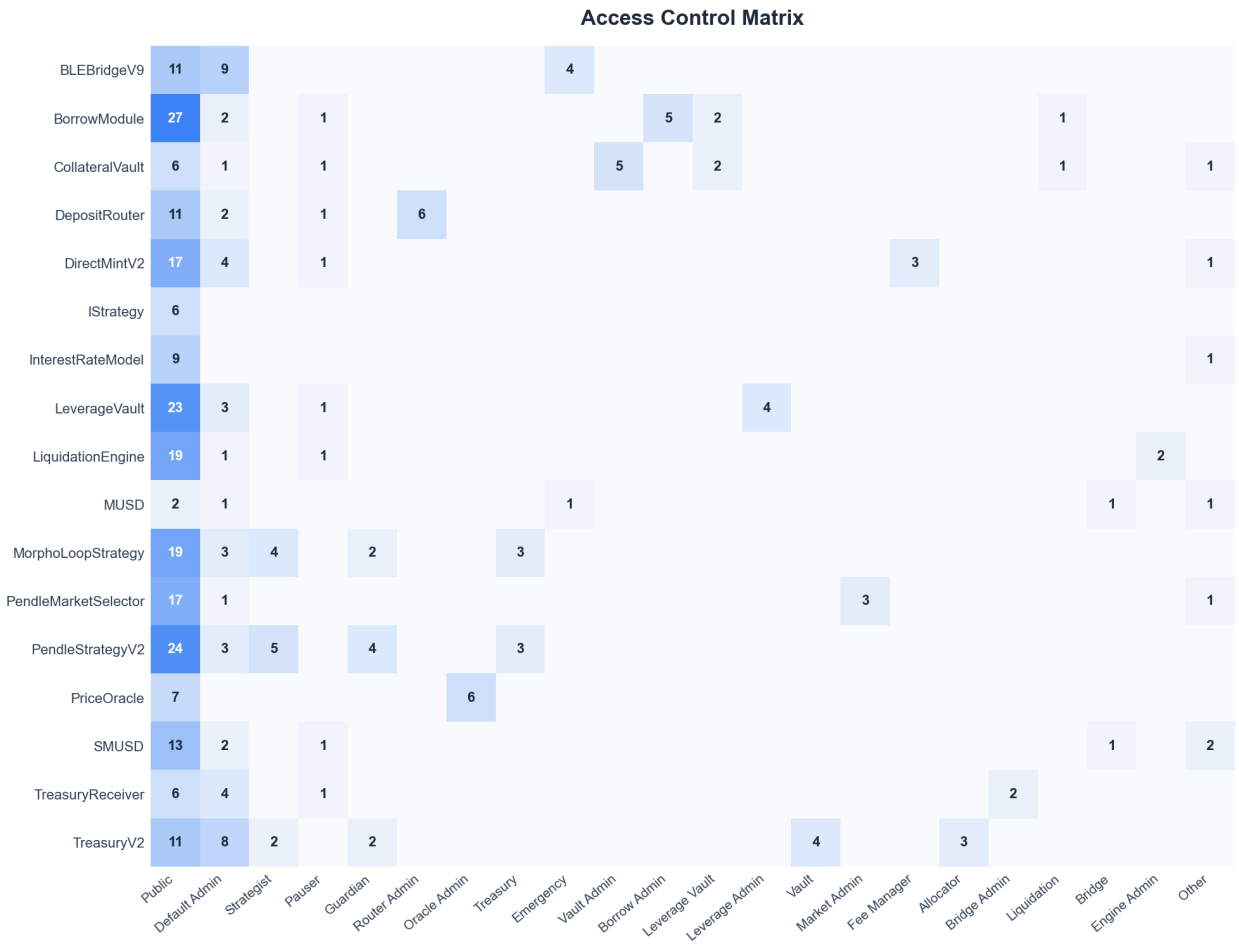
5.3 Source Lines & Risk



5.4 Capabilities

Module	Functions	Public	Restricted	Key Roles
BLEBridgeV9	24	11	13	DEFAULT_ADMIN, EMERGENCY
BorrowModule	38	27	11	BORROW_ADMIN, LEVERAGE_VAULT, DEFAULT_ADMIN
CollateralVault	17	6	11	VAULT_ADMIN, LEVERAGE_VAULT, BORROW_MODULE
DepositRouter	20	11	9	ROUTER_ADMIN, DEFAULT_ADMIN, PAUSER
DirectMintV2	26	17	9	DEFAULT_ADMIN, FEE_MANAGER, MINTER
IStrategy	6	6	0	-
InterestRateModel	10	9	1	RATE_ADMIN
LeverageVault	31	23	8	LEVERAGE_ADMIN, DEFAULT_ADMIN, PAUSER
LiquidationEngine	23	19	4	ENGINE_ADMIN, PAUSER, DEFAULT_ADMIN
MUSD	6	2	4	COMPLIANCE, BRIDGE, EMERGENCY
MorphoLoopStrategy	31	19	12	STRATEGIST, TREASURY, DEFAULT_ADMIN
PendleMarketSelector	22	17	5	MARKET_ADMIN, PARAMS_ADMIN, DEFAULT_ADMIN

Module	Functions	Public	Restricted	Key Roles
PendleStrategyV2	39	24	15	STRATEGIST, GUARDIAN, TREASURY
PriceOracle	13	7	6	ORACLE_ADMIN
SMUSD	19	13	6	DEFAULT_ADMIN, YIELD_MANAGER, INTEREST_ROUTER
TreasuryReceiver	13	6	7	DEFAULT_ADMIN, BRIDGE_ADMIN, PAUSER
TreasuryV2	30	11	19	DEFAULT_ADMIN, VAULT, ALLOCATOR



5.5 Dependencies / External Imports

Import Path	Version
@openzeppelin/contracts-upgradeable/access/AccessControlUpgradeable.sol	^5.0.0

Import Path	Version
@openzeppelin/contracts-upgradeable/proxy/utils/UUPSUpgradeable.sol	^5.0.0
@openzeppelin/contracts-upgradeable/utils/PausableUpgradeable.sol	^5.0.0
@openzeppelin/contracts-upgradeable/utils/ReentrancyGuardUpgradeable.sol	^5.0.0
@openzeppelin/contracts/access/AccessControl.sol	^5.0.0
@openzeppelin/contracts/token/ERC20/ERC20.sol	^5.0.0
@openzeppelin/contracts/token/ERC20/IERC20.sol	^5.0.0
@openzeppelin/contracts/token/ERC20/extensions/ERC4626.sol	^5.0.0
@openzeppelin/contracts/token/ERC20/extensions/IERC20Metadata.sol	^5.0.0
@openzeppelin/contracts/token/ERC20/utils/SafeERC20.sol	^5.0.0
@openzeppelin/contracts/utils/Pausable.sol	^5.0.0
@openzeppelin/contracts/utils/ReentrancyGuard.sol	^5.0.0
@openzeppelin/contracts/utils/cryptography/ECDSA.sol	^5.0.0
@openzeppelin/contracts/utils/cryptography/MessageHashUtils.sol	^5.0.0
@openzeppelin/contracts/utils/math/Math.sol	^5.0.0

5.6 Source Units in Scope

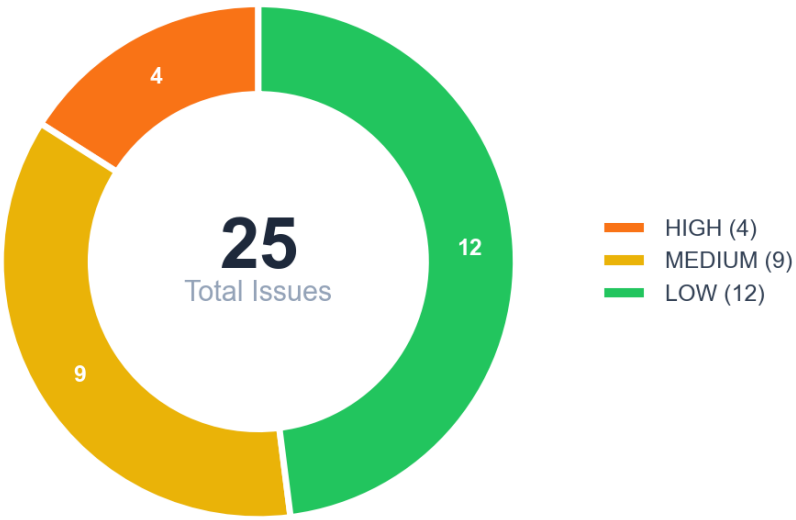
File	Instructions	Functions	Lines	nSLOC	Comments
DepositRouter.sol	16	21	422	229	126
PendleMarketSelector.sol	17	26	534	266	170
PriceOracle.sol	11	13	256	168	50
BLEBridgeV9.sol	20	28	475	281	122
TreasuryReceiver.sol	13	13	295	156	95
CollateralVault.sol	12	17	299	169	82
InterestRateModel.sol	2	10	276	142	96
LeverageVault.sol	28	36	770	467	168
DirectMintV2.sol	25	26	324	195	70
MUSD.sol	6	7	104	70	18

File	Instructions	Functions	Lines	nSLOC	Comments
LiquidationEngine.sol	21	23	274	150	79
TreasuryV2.sol	23	34	1000	534	291
SMUSD.sol	19	23	329	167	105
BorrowModule.sol	33	46	830	481	204
MorphoLoopStrategy.sol	16	36	811	440	235
PendleStrategyV2.sol	31	46	830	500	187
IStrategy.sol	6	6	40	9	25
Total	**299**	**411**	**7869**	**4424**	**2123**

6. Scope of Work

The Minted team has provided the Solidity smart contracts for their DeFi lending protocol. The audit focuses on validating security, compliance, and robustness.

6.1 Findings Overview



No	Title	Severity	Status
6.2.1	Vault Insolvency From globalTotalAssets() vs Local Balance Mismatch	HIGH	FIXED
6.2.2	Silent Treasury Fallback in globalTotalAssets() Enables Share Price Manipulation	HIGH	FIXED
6.2.3	Canton Share Sync Rate Limit Compounds Exponentially	HIGH	FIXED
6.2.4	First Canton Sync Bypass Accepts Arbitrary Share Value at Zero Supply	HIGH	FIXED
6.2.5	Wormhole Relayer and Token Bridge Fee Mismatch Causes Deposit Failures	MEDIUM	FIXED

No	Title	Severity	Status
6.2.6	_fullDeleverage() May Fail to Fully Unwind Due to Interest-Driven LTV Drift	MEDIUM	ACKNOWLEDGED
6.2.7	No Bad Debt Handling Creates Permanent Zombie Debt	MEDIUM	FIXED
6.2.8	emergencyClosePosition() Reverts When Swap Fails Trapping User Funds	MEDIUM	FIXED
6.2.9	Blacklist Does Not Check msg.sender in transferFrom	MEDIUM	FIXED
6.2.10	TVL Unit Mismatch Between SY-Denominated Value and USD Threshold	MEDIUM	FIXED
6.2.11	lastLnImpliedRate Is a Spot Value That Can Be Manipulated	MEDIUM	FIXED
6.2.12	Deposit During Rollover Window Orphans Old PT and Corrupts ptBalance	MEDIUM	FIXED
6.2.13	distributeYield() and receiveInterest() Are Sandwich-Attackable	MEDIUM	FIXED
6.2.14	emergencyWithdraw Can Drain USDC Backing accumulatedFees	LOW	FIXED
6.2.15	rebalance() Strategy totalValue() Calls Not Wrapped in try/catch, Causing DoS	LOW	FIXED
6.2.16	setReserveBps() Lacks Allocation Consistency Validation	LOW	FIXED
6.2.17	emergencyWithdrawAll() Does Not Pause the Contract	LOW	FIXED
6.2.18	ETH Refund Reverts Entire Deposit for Smart Contract Callers	LOW	ACKNOWLEDGED
6.2.19	setParams() Does Not Accrue Interest Before Rate Change	LOW	FIXED
6.2.20	minSupplyRateRequired Is Set But Never Enforced	LOW	ACKNOWLEDGED
6.2.21	withdraw() Deducts Full Principal Regardless of Actual Amount Freed	LOW	ACKNOWLEDGED
6.2.22	No Oracle Cardinality Check Causes TWAP Query to Revert	LOW	FIXED
6.2.23	emergencyWithdraw Emits Wrong ptBalance Value in Event	LOW	ACKNOWLEDGED
6.2.24	resetLastKnownPrice() Skips Staleness and Circuit Breaker Checks	LOW	FIXED
6.2.25	Legacy withdraw() Does Not Update lastRecordedValue, Causing Fee Drift	LOW	FIXED

6.2 Manual and Automated Vulnerability Test

6.2.1 Vault Insolvency From globalTotalAssets() vs Local Balance Mismatch

HIGH FIXED

DESCRIPTION

The SMUSD vault computes share-to-asset conversions usingglobalTotalAssets(), which derives fromTreasury.totalValue() * 1e12and includes all capital deployed across yield strategies. However, the actual withdrawal mechanism inherited from OpenZeppelin's ERC4626 can only transfer mUSD from the vault's localbalanceOf(address(this)).

When the treasury's total value exceeds the vault's local mUSD balance (the normal operating state since the treasury deploys capital into strategies), the vault becomes structurally insolvent. It promises more mUSD per share than it holds. Users callingredeem()get a revert because the vault tries to transfer more than its local balance. This creates a first-come-first-served bank run where the first withdrawer drains the vault and remaining users are locked out.

Additionally,maxWithdraw()andmaxRedeem()are not overridden and return values based on global pricing, violating the ERC-4626 spec requirement that these functions must return values that do not cause a revert.

CODE LOCATION

File:contracts/SMUSD.sol, Lines 306-313

contracts/SMUSD.sol (Lines 306-313)

```
function _convertToAssets(uint256 shares, Math.Rounding rounding) internal view
override returns (uint256) {
uint256 totalShares = globalTotalShares();
return shares.mulDiv(globalTotalAssets() + 1, totalShares + 10 **
_decimalsOffset(), rounding);
// ^^^^^^^^^^^^^^^^^ Treasury.totalValue * 1e12 (global)
}
function _withdraw(...) internal virtual {
_burn(owner, shares);
SafeERC20.safeTransfer(IERC20(asset()), to, assets);
// ^^^^^^ Can only send local balance
}
```

RECOMMENDATION

Override_withdraw()to pull from treasury when local balance is insufficient:

Also overridemaxWithdraw()andmaxRedeem()to cap at local liquidity.

```
function _withdraw(address caller, address receiver, address owner, uint256
assets, uint256 shares) internal override {
```

```
uint256 localBal = IERC20(asset()).balanceOf(address(this));
if (assets > localBal) {
uint256 deficit = assets - localBal;
ITreasury(treasury).withdrawTo(address(this), deficit);
}
super._withdraw(caller, receiver, owner, assets, shares);
}
```

6.2.2 Silent Treasury Fallback in globalTotalAssets() Enables Share Price Manipulation

HIGH FIXED

DESCRIPTION

TheglobalTotalAssets()function uses atry/catchto queryITreasury(treasury).totalValue(). If the treasury call reverts for any reason, the function silently falls back tototalAssets()(the local mUSD balance). This creates a massive share price discontinuity with no event emission, no circuit breaker, and no cached fallback value.

When the treasury manages 5M USDC but the vault only holds 500K mUSD locally, a treasury failure causesglobalTotalAssets()to drop from 5M to 500K (a 10x drop). An attacker who observes or triggers the treasury failure deposits at the crashed price, receiving roughly 10x more shares than normal. After the treasury recovers and the 24-hour cooldown expires, the attackerredeems at the restored price, extracting value from existing depositors.

CODE LOCATION

File:contracts/SMUSD.sol, Lines 251-265

contracts/SMUSD.sol (Lines 251-265)

```
function globalTotalAssets() public view returns (uint256) {
if (treasury == address(0)) {
return totalAssets();
}
try ITreasury(treasury).totalValue() returns (uint256 usdcValue) {
return usdcValue * 1e12;
} catch {
return totalAssets(); // Silent fallback, no event, no cache
}
}
```

RECOMMENDATION

Remove the silent fallback and let the revert propagate:

If availability is needed, use a cached last-known-good value with a staleness check instead of falling back to the local balance.

```
function globalTotalAssets() public view returns (uint256) {
if (treasury == address(0)) return totalAssets();
uint256 usdcValue = ITreasury(treasury).totalValue();
return usdcValue * 1e12;
}
```

6.2.3 Canton Share Sync Rate Limit Compounds Exponentially

HIGH FIXED

DESCRIPTION

ThesyncCantonShares()function has a per-sync rate limit of 5% change per hour. While thisblocks sudden large changes in a single transaction, the rate limit compounds across consecutive syncs because each sync's cap is computed against the current (already inflated)cantonTotalShares. A compromised BRIDGE_ROLE can inflate canton shares by 3.18x in 24 hours, 10.2x in 48 hours, and 129x in 100 hours.

Since_convertToAssets()usesglobalTotalShares()(Ethereum shares + Canton shares) as the denominator, inflated canton shares dilute every Ethereum depositor's per-share asset claim. There is no cumulative cap, no absolute maximum ratio between Canton and Ethereum shares, and no governance escalation threshold.

CODE LOCATION

File:contracts/SMUSD.sol, Lines 227-230

contracts/SMUSD.sol (Lines 227-230)

```
uint256 maxIncrease = (cantonTotalShares * (10000 + MAX_SHARE_CHANGE_BPS)) /
10000;
uint256 maxDecrease = (cantonTotalShares * (10000 - MAX_SHARE_CHANGE_BPS)) /
10000;
require(_cantonShares <= maxIncrease, "SHARE_INCREASE_TOO_LARGE");
require(_cantonShares >= maxDecrease, "SHARE_DECREASE_TOO_LARGE");
```

RECOMMENDATION

Add a cumulative change cap over a rolling time window:

Also add an absolute maximum ratio between Canton and Ethereum shares.

```
uint256 public cantonSharesAtWindowStart;
uint256 public windowStartTime;
uint256 public constant WINDOW_DURATION = 24 hours;
```

```
uint256 public constant MAX_CUMULATIVE_CHANGE_BPS = 1500; // 15% per 24h
// In syncCantonShares():
if (block.timestamp >= windowStartTime + WINDOW_DURATION) {
cantonSharesAtWindowStart = cantonTotalShares;
windowStartTime = block.timestamp;
}
uint256 maxCumulative = (cantonSharesAtWindowStart * (10000 +
MAX_CUMULATIVE_CHANGE_BPS)) / 10000;
require(_cantonShares <= maxCumulative, "CUMULATIVE_INCREASE_EXCEEDED");
```

6.2.4 First Canton Sync Bypass Accepts Arbitrary Share Value at Zero Supply

HIGH FIXED

DESCRIPTION

The first-sync guard insyncCantonShares()is designed to cap initial Canton shares at 2x Ethereum shares. But whentotalSupply() == 0(no Ethereum deposits yet), the guard degrades torequire(_cantonShares <= _cantonShares), which is always true. This is a tautological check that provides zero protection.

A BRIDGE_ROLE holder can inject any arbitrary value ascantonTotalShareson a freshly deployed vault. Setting it to a huge value either bricks the vault permanently (all deposits revert with overflow in_convertToShares) or enables phantom Canton shares to steal over 99% of all future yield from Ethereum depositors.

The irrecoverable nature comes from the 5% per-hour rate limit on subsequent syncs. Reducing from a massive value to a sane one would take hundreds of hours.

CODE LOCATION

File:contracts/SMUSD.sol, Lines 222-226

contracts/SMUSD.sol (Lines 222-226)

```
if (cantonTotalShares == 0) {
uint256 ethShares = totalSupply();
uint256 maxInitialShares = ethShares > 0 ? ethShares * 2 : _cantonShares;
// ^^^^^^^^^^^
// When ethShares == 0: maxInitialShares = _cantonShares (the INPUT)
require(_cantonShares <= maxInitialShares, "INITIAL_SHARES_TOO_LARGE");
// becomes: require(_cantonShares <= _cantonShares) -> ALWAYS TRUE
}
```

RECOMMENDATION

Require Ethereum deposits to exist before the first Canton sync:

```
if (cantonTotalShares == 0) {
uint256 ethShares = totalSupply();
require(ethShares > 0, "NO_ETH_DEPOSITS_YET");
require(_cantonShares <= ethShares * 2, "INITIAL_SHARES_TOO_LARGE");
}
```

6.2.5 Wormhole Relayer and Token Bridge Fee Mismatch Causes Deposit Failures

MEDIUM FIXED

DESCRIPTION

quoteBridgeCost() queries the Wormhole Automatic Relayer for a cross-chain delivery price quote, but _deposit() sends the quoted ETH to the Wormhole Token Bridge via transferTokensWithPayload(). These are two completely different Wormhole services with incompatible fee structures.

The Token Bridge internally forwards the entire msg.value to the Wormhole Core publishMessage() function, which enforces require(msg.value == messageFee()). Since messageFee() is typically 0 on all EVM chains and the relayer quote is much greater than 0, every deposit reverts. This makes the contract permanently non-functional.

Even in a lenient Wormhole Core scenario, the user ETH would be donated to the Wormhole Core contract with no recovery, while the Relayer receives nothing and no automatic delivery occurs.

```
function quoteBridgeCost() public view returns (uint256 nativeCost) {
(nativeCost, ) = wormholeRelayer.quoteEVMDeliveryPrice(
ETHEREUM_CHAIN_ID, 0, GAS_LIMIT
);
}
sequence = tokenBridge.transferTokensWithPayload{value: bridgeCost}(
address(usdc), netAmount, ETHEREUM_CHAIN_ID,
treasuryReceiverBytes, _nonce, recipientPayload
);
```

RECOMMENDATION

Replace the relayer fee quote with the Token Bridge message fee. Change quoteBridgeCost() to query wormhole().messageFee() from the Token Bridge instead of quoteEVMDeliveryPrice() from the Relayer. If automatic cross-chain delivery is intended, use the Relayer service for the actual transfer as well, not the Token Bridge.

6.2.6 _fullDeleverage() May Fail to Fully Unwind Due to Interest-Driven LTV Drift

MEDIUM ACKNOWLEDGED

DESCRIPTION

_fullDeleverage() is capped at MAX_LOOPS * 2 = 10 iterations. In Morpho Blue, collateral does not earn interest, but borrow debt accrues continuously. Over time, the effective LTV drifts upward toward LLTV as debt grows while collateral stays constant.

When the effective LTV exceeds safeLtv (targetLtvBps - safetyBufferBps), _maxWithdrawable() returns 0 for every iteration after the initial USDC balance is consumed. The loop produces zero progress for all remaining iterations and exits silently with the position partially unwound.

withdrawAll() then sets totalPrincipal = 0, creating an accounting desync where the strategy believes it is empty while funds remain trapped in Morpho with accruing debt and no recovery mechanism. emergencyDeleverage() uses the same _fullDeleverage() and also stalls.

```
for (uint256 i = 0; i < MAX_LOOPS * 2; i++) {
// ...
uint256 maxWithdraw = _maxWithdrawable();
// When LTV > safeLtv, maxWithdraw == 0 and loop is a no-op
}
```

RECOMMENDATION

Increase the iteration limit. Add a fallback that uses LLTV-based withdrawal when safeLtv is too restrictive. Add a require check after the loop to verify the position is fully unwound (borrowShares == 0 and collateral == 0) before setting totalPrincipal to 0.

6.2.7 No Bad Debt Handling Creates Permanent Zombie Debt

MEDIUM FIXED

DESCRIPTION

When a position becomes severely undercollateralized (collateral value less than total debt), the liquidate() function seizes all available collateral but can only reduce debt by the equivalent collateral value minus penalty. The remaining debt is never cleared and has no recovery mechanism anywhere in the protocol.

The protocol has no clearBadDebt(), socializeLoss(), or any admin debt-clearing function in LiquidationEngine or BorrowModule. The zombie debt remains in totalBorrows permanently, accruing interest on debt backed by nothing, inflating the utilization rate, and raising interest rates for all other borrowers.

RECOMMENDATION

Add a clearBadDebt(address borrower) function callable by an authorized role that writes off remaining debt when a position has zero collateral. Socialize the loss across the protocol (e.g., through a reserve fund or pro-rata deduction from suppliers). This matches the approach used by Aave V3 (handleBadDebt), Compound, and Liquity.

6.2.8 emergencyClosePosition() Reverts When Swap Fails Trapping User Funds

MEDIUM FIXED

DESCRIPTION

emergencyClosePosition() is the last-resort recovery function for stuck leveraged positions. It calls _swapCollateralToMusd() which reverts on swap failure. If the Uniswap pool has insufficient liquidity, the token is paused or blacklisted, or the pool does not exist, the emergency close itself reverts, defeating its purpose and permanently trapping user funds.

There is an inconsistency in error handling: _swapMusdToCollateral() uses try/catch on swapRouter.exactInputSingle(), and borrowModule.repayFor() in the emergency function is wrapped in try/catch. But _swapCollateralToMusd() is called directly without try/catch, causing the entire emergency close to revert on swap failure.

No alternative recovery path exists. emergencyWithdraw() only affects tokens held in LeverageVault (which holds 0 since collateral goes to CollateralVault). CollateralVault has no emergencyWithdraw() function.

```
uint256 musdReceived = _swapCollateralToMusd(collateralToken,
totalCollateralInVault);
// If swap reverts, entire emergency close reverts
musdReceived = swapRouter.exactInputSingle(
ISwapRouter.ExactInputSingleParams({ ... })
);
require(musdReceived > 0, "SWAP_RETURNED_ZERO");
```

RECOMMENDATION

Wrap the _swapCollateralToMusd() call in emergencyClosePosition() with try/catch. If the swap fails,skip debt repayment and return the raw collateral to the user. Accept the protocol bad debt as the cost of the emergency recovery.

6.2.9 Blacklist Does Not Check msg.sender in transferFrom

MEDIUM FIXED

DESCRIPTION

The_update()override in MUSD.sol only checksfromandtoagainst the blacklist. It does not checkmsg.sender. In atransferFromcall, the operator (msg.sender) can be a different address thanfromorto. A blacklisted operator can still calltransferFromto move tokens between non-blacklisted addresses, as long as they have an existing approval. This defeats the purpose of the blacklist for compliance, since a sanctioned entity can continue to facilitate token movements.

OpenZeppelin's ERC20transferFromflow calls_spendAllowancethen_update. Since_updateonly checksfromandto, themsg.senderoperator is never validated against the blacklist.

CODE LOCATION

File:contracts/MUSD.sol, Line 90

contracts/MUSD.sol (Lines 90-90)

```
function _update(address from, address to, uint256 value) internal override {
if (from != address(0)) {
require(!blacklisted[from], "BLACKLISTED");
}
if (to != address(0)) {
require(!blacklisted[to], "BLACKLISTED");
}
super._update(from, to, value);
}
```

RECOMMENDATION

OverridetransferFromor add amsg.sendercheck inside_updatefor non-mint transfers:

```
function _update(address from, address to, uint256 value) internal override {
if (from != address(0)) {
require(!blacklisted[from], "BLACKLISTED");
require(!blacklisted[msg.sender], "BLACKLISTED_OPERATOR");
}
if (to != address(0)) {
require(!blacklisted[to], "BLACKLISTED");
}
```

```
super._update(from, to, value);
}
```

6.2.10 TVL Unit Mismatch Between SY-Denominated Value and USD Threshold

MEDIUM FIXED

DESCRIPTION

ThegetValidMarkets()function comparestvlSy(a value denominated in the SY token's native units) againstminTvlUsd(a threshold denominated in USD). For example, if the SY token is wstETH (18 decimals, price ~$2000), a market with 500 wstETH ($1M) would havetvlSy = 500e18. IfminTvlUsdis set to1_000_000e6($1M in 6-decimal USD), the comparisontvlSy >= minTvlUsdbecomes500e18 >= 1_000_000e6, which passes even though both represent $1M. The comparison is meaningless because the units are incompatible.

This can cause markets with insufficient real TVL to pass the filter (false positives) or adequate markets to be rejected (false negatives), depending on the SY token's price and decimals.

CODE LOCATION

File:contracts/PendleMarketSelector.sol, Line 208

contracts/PendleMarketSelector.sol (Lines 208-208)

```
if (tvlSy < minTvlUsd) continue; // Comparing SY units to USD units
(int128 totalPt, int128 totalSy, , , ,) = IPendleMarket(market)._storage();
uint256 tvlSy = uint256(uint128(totalSy));
```

RECOMMENDATION

ConverttvlSyto a USD value before comparing againstminTvlUsd. Use the Pendle oracle or a Chainlink feed to get the SY token price:

```
uint256 syPriceUsd = IPendleOracle(PENDLE_ORACLE).getSyToAssetRate(market) *
assetPriceUsd / 1e18;
uint256 tvlUsd = tvlSy * syPriceUsd / (10 ** syDecimals);
if (tvlUsd < minTvlUsd) continue;
```

6.2.11 lastLnImpliedRate Is a Spot Value That Can Be Manipulated

MEDIUM FIXED

DESCRIPTION

The_getMarketInfo()function readslastLnImpliedRatedirectly from the market's_storage()struct. This is a spot value that reflects the most recent trade and can be manipulated via a large swap within the same block. An attacker can inflate or deflate the implied rate by making a large trade right beforeselectBestMarket()is called, causing the selector to pick a suboptimal market. The attacker then profits from this mispricing by trading in the selected market.

The contract already uses the Pendle TWAP oracle for thegetPtToSyRatecall, showing awareness of manipulation risks. But the implied rate scoring bypasses this protection entirely by using the raw spot value.

CODE LOCATION

File:contracts/PendleMarketSelector.sol, Lines 315-330

contracts/PendleMarketSelector.sol (Lines 315-330)

```
function _getMarketInfo(address market) internal view returns (MarketInfo memory
info) {
// ...
(, , uint96 lastLnImpliedRate, , ,) = IPendleMarket(market)._storage();
info.impliedRate = uint256(lastLnImpliedRate); // Spot value, manipulable
info.impliedAPY = _lnRateToAPY(lastLnImpliedRate);
info.score = _calculateScore(info);
// ...
}
```

RECOMMENDATION

Use a TWAP-based implied rate instead of the spotlastLnImpliedRate. Pendle providesgetOracleState()which returns TWAP-based rates. Alternatively, use a time-weighted observation from the market's oracle:

```
uint256 twapRate = IPendleOracle(PENDLE_ORACLE).getImpliedRateTWAP(market,
TWAP_DURATION);
info.impliedRate = twapRate;
info.impliedAPY = _lnRateToAPY(uint96(twapRate));
```

6.2.12 Deposit During Rollover Window Orphans Old PT and Corrupts ptBalance

MEDIUM FIXED

DESCRIPTION

When_shouldRollover()returns true during adeposit()call, the function calls_selectNewMarket()which switches all market pointers to a new Pendle market without first redeeming the existing PT tokens from the old market. This permanently orphans the old PT tokens and corrupts theptBalanceaccounting variable.

The correct rollover implementation inrollToNewMarket()first redeems old PT, then selects the new market, then re-deposits. Thedeposit()function skips the redemption step entirely.

After the auto-rollover,ptBalancecontains the sum of old orphaned PT count plus new PT count, but the contract only holds new PT. This causeswithdrawAll()to revert (tries to redeem more PT than exists),totalValue()to report inflated NAV, and individual withdrawals to fail once cumulative withdrawals exceed actual PT held.

CODE LOCATION

File:contracts/strategies/PendleStrategyV2.sol, Lines 368-419

contracts/strategies/PendleStrategyV2.sol (Lines 368-419)

```
function deposit(uint256 amount) external override nonReentrant whenNotPaused
onlyRole(TREASURY_ROLE) returns (uint256 deposited) {
if (currentMarket == address(0) || _shouldRollover()) {
_selectNewMarket(); // Switches market pointers immediately
// Missing: _redeemPt(ptBalance) to redeem old PT first
}
// ... swap USDC to new PT ...
ptBalance += netPtOut; // ptBalance = old_count + new_count (corrupted)
}
function rollToNewMarket() external onlyRole(TREASURY_ROLE) {
if (ptBalance > 0) {
usdcRecovered = _redeemPt(ptBalance); // Step 1: Redeem old PT
}
_selectNewMarket(); // Step 2: Switch market
if (usdcRecovered > 0) {
_depositToCurrentMarket(usdcRecovered);// Step 3: Re-deposit
}
}
```

RECOMMENDATION

Redeem old PT before selecting the new market indeposit():

Or remove auto-rollover fromdeposit()entirely and require an explicitrollToNewMarket()call.

```
if (currentMarket == address(0) || _shouldRollover()) {
if (ptBalance > 0) {
uint256 recovered = _redeemPt(ptBalance);
}
```

```
_selectNewMarket();
}
```

6.2.13 distributeYield() and receiveInterest() Are Sandwich-Attackable

MEDIUM FIXED

DESCRIPTION

BothdistributeYield()andreceiveInterest()transfer mUSD lump-sum into the SMUSD vault, instantly inflatingtotalAssets()and the share price for all holders. Shares become yield-eligible immediately upon deposit. The 24-hour cooldown blocks atomic same-block sandwich attacks but does not prevent strategic yield timing attacks.

An attacker who deposits a large amount 25 hours before a predictable yield distribution captures a proportional share of the yield. Since yield distributions are permissioned (role-gated), their timing is predictable. A whale depositing equal to the vault total captures 50% of the yield despite holding shares for only 25 hours, while a long-term staker who held for 30 days receives only 50%.

CODE LOCATION

File:contracts/SMUSD.sol, Lines 136-165

contracts/SMUSD.sol (Lines 136-165)

```
function distributeYield(uint256 amount) external onlyRole(YIELD_MANAGER_ROLE) {
require(amount > 0, "ZERO_AMOUNT");
uint256 maxYield = (totalAssets() * MAX_YIELD_BPS) / 10000;
require(amount <= maxYield, "YIELD_TOO_LARGE");
IERC20(asset()).safeTransferFrom(msg.sender, address(this), amount);
// Instantly increases totalAssets() for all holders
emit YieldDistributed(amount);
}
function receiveInterest(uint256 amount) external onlyRole(INTEREST_ROUTER_ROLE) {
require(amount > 0, "ZERO_AMOUNT");
IERC20(asset()).safeTransferFrom(msg.sender, address(this), amount);
emit InterestReceived(amount);
}
```

RECOMMENDATION

Implement yield streaming instead of lump-sum distribution:

This spreads yield over time so short-term depositors cannot capture a disproportionate share.

```
uint256 public yieldVestingEnd;
uint256 public unvestedYield;
function distributeYield(uint256 amount) external onlyRole(YIELD_MANAGER_ROLE) {
```

```
unvestedYield += amount;
yieldVestingEnd = block.timestamp + 48 hours;
IERC20(asset()).safeTransferFrom(msg.sender, address(this), amount);
}
function totalAssets() public view override returns (uint256) {
uint256 vested = _vestedAmount();
return IERC20(asset()).balanceOf(address(this)) - (unvestedYield - vested);
}
```

6.2.14 emergencyWithdraw Can Drain USDC Backing accumulatedFees

LOW FIXED

DESCRIPTION

emergencyWithdraw() allows DEFAULT_ADMIN_ROLE to withdraw any ERC20 token including USDC, but it never reads or writes accumulatedFees. There is also no admin setter for accumulatedFees anywhere in the contract. After an emergency withdrawal of USDC, the accumulatedFees state variable still reflects the old balance, creating a permanent accounting desync.

When withdrawFees() is called after such a drain, it reads the stale accumulatedFees value, sets it to 0, and attempts to transfer that amount. Since the actual USDC balance is insufficient, the transfer reverts. This permanently blocks fee collection with no recovery path.

```
function emergencyWithdraw(address token, address to, uint256 amount) external
onlyRole(DEFAULT_ADMIN_ROLE) {
if (to == address(0)) revert InvalidAddress();
if (token == address(0)) {
(bool success, ) = to.call{value: amount}("");
if (!success) revert TransferFailed();
} else {
IERC20(token).safeTransfer(to, amount);
}
}
function withdrawFees(address to) external onlyRole(ROUTER_ADMIN_ROLE) {
if (to == address(0)) revert InvalidAddress();
uint256 amount = accumulatedFees;
accumulatedFees = 0;
usdc.safeTransfer(to, amount);
emit FeesWithdrawn(to, amount);
}
```

RECOMMENDATION

In emergencyWithdraw(), when the token is USDC, reduce accumulatedFees by the withdrawn amount (capping at zero). This keeps the accounting variable in sync with the actual contract balance.

6.2.15 rebalance() Strategy totalValue() Calls Not Wrapped in try/catch, Causing DoS

LOW FIXED

DESCRIPTION

EveryIStrategy.totalValue()call in TreasuryV2 is wrapped intry/catchexcept the two insiderebalance()at Lines 842 and 870. If any active strategy'stotalValue()reverts (due to an external protocol pause, upgrade, or bug), the entirerebalance()transaction reverts. This blocks rebalancing for all strategies, not just the failing one.

The inconsistency is clear: thetotalValue()view function (Line 224),getCurrentAllocations()(Line 297),_withdrawFromStrategies()loops (Lines 567, 587), andremoveStrategy()(Line 753) all havetry/catch. Onlyrebalance()is missing it.

The three integrated DeFi protocols (Pendle, Morpho, Sky) all have known pause and upgrade mechanisms that could trigger this.

CODE LOCATION

File:contracts/TreasuryV2.sol, Lines 842 and 870

contracts/TreasuryV2.sol (Lines 842-842)

```
// Pass 1 - Line 842: withdraw from over-allocated strategies
uint256 currentValue = IStrategy(strat).totalValue(); // No try/catch
// Pass 2 - Line 870: deposit to under-allocated strategies
uint256 currentValue = IStrategy(strat).totalValue(); // No try/catch
```

RECOMMENDATION

Wrap both calls intry/catchconsistent with the rest of the contract:

```
uint256 currentValue;
try IStrategy(strat).totalValue() returns (uint256 val) {
currentValue = val;
} catch {
emit RebalanceSkipped(strat);
continue;
}
```

6.2.16 setReserveBps() Lacks Allocation Consistency Validation

LOW FIXED

DESCRIPTION

ThesetReserveBps()function only validates that the new value is at most 3000 (30%) but does not check whetherreserveBpsplus the sum of all active strategytargetBpsexceeds 10000 (100%). BothaddStrategy()andupdateStrategy()enforce this invariant, butsetReserveBps()bypasses it.

When the invariant is broken (e.g., strategies total 9000 bps and reserve is set to 3000 bps = 12000 bps total),addStrategy()andupdateStrategy()revert withTotalAllocationInvalid, locking strategy management. Therebalance()function also malfunctions because it calculates target reserve as 30% of total, but actual reserve is only 10%, so no funds are deployed to under-allocated strategies. New deposits route 30% to reserve instead of the intended 10%, reducing depositor yield.

CODE LOCATION

File:contracts/TreasuryV2.sol, Line 957

contracts/TreasuryV2.sol (Lines 957-957)

```
function setReserveBps(uint256 _reserveBps) external onlyRole(DEFAULT_ADMIN_ROLE)
{
require(_reserveBps <= 3000, "Reserve too high");
uint256 oldBps = reserveBps;
reserveBps = _reserveBps;
emit ReserveBpsUpdated(oldBps, _reserveBps);
// Missing: validate reserveBps + sum(strategies[i].targetBps) <= 10000
}
```

RECOMMENDATION

Add the same allocation consistency check used byaddStrategy()andupdateStrategy():

```
function setReserveBps(uint256 _reserveBps) external onlyRole(DEFAULT_ADMIN_ROLE)
{
require(_reserveBps <= 3000, "Reserve too high");
uint256 totalTarget = _reserveBps;
for (uint256 i = 0; i < strategies.length; i++) {
if (strategies[i].active) totalTarget += strategies[i].targetBps;
}
require(totalTarget <= BPS, "Total allocation exceeds 100%");
uint256 oldBps = reserveBps;
reserveBps = _reserveBps;
emit ReserveBpsUpdated(oldBps, _reserveBps);
}
```

6.2.17 emergencyWithdrawAll() Does Not Pause the Contract

LOW FIXED

DESCRIPTION

TheemergencyWithdrawAll()function pulls all funds from strategies into the reserve but does not pause the contract or deactivate strategies. Between the emergency withdrawal and a separate manualpause()call, a vault deposit can arrive. Since the contract is unpaused and strategies remain active,depositFromVault()executes and_autoAllocate()re-deploys funds to all active strategies, including the potentially compromised one that triggered the emergency.

This effectively nullifies the emergency action. At minimum a 1-block race window exists between emergency withdrawal and manual pause.

CODE LOCATION

File:contracts/TreasuryV2.sol, Lines 902-916

contracts/TreasuryV2.sol (Lines 902-916)

```
function emergencyWithdrawAll() external onlyRole(GUARDIAN_ROLE) {
for (uint256 i = 0; i < strategies.length; i++) {
if (strategies[i].active) {
address strategyAddr = strategies[i].strategy;
try IStrategy(strategyAddr).withdrawAll() {}
catch (bytes memory reason) {
emit StrategyWithdrawFailed(strategyAddr, 0, reason);
}
}
}
emit EmergencyWithdraw(reserveBalance());
// Missing: _pause();
}
```

RECOMMENDATION

Add_pause()at the end ofemergencyWithdrawAll():

```
function emergencyWithdrawAll() external onlyRole(GUARDIAN_ROLE) {
for (uint256 i = 0; i < strategies.length; i++) {
if (strategies[i].active) {
address strategyAddr = strategies[i].strategy;
try IStrategy(strategyAddr).withdrawAll() {}
catch (bytes memory reason) {
emit StrategyWithdrawFailed(strategyAddr, 0, reason);
}
}
}
lastRecordedValue = totalValue();
```

```
_pause();
emit EmergencyWithdraw(reserveBalance());
}
```

6.2.18 ETH Refund Reverts Entire Deposit for Smart Contract Callers

LOW ACKNOWLEDGED

DESCRIPTION

The _deposit() function uses a push-based ETH refund at lines 412-415. When msg.sender is a smart contract without a receive() or fallback() function and sends even 1 wei more than bridgeCost, the refund call fails and the entire deposit transaction reverts via revert TransferFailed().

The bridge cost quote from quoteBridgeCost() can change between blocks. If the cost decreases between when the caller queries it and when the transaction executes, msg.value > bridgeCost becomes true unexpectedly. Smart contract wallets, automation bots, and aggregator contracts that lack a receive() function are permanently blocked from depositing in this scenario.

```
if (msg.value > bridgeCost) {
(bool success, ) = msg.sender.call{value: msg.value - bridgeCost}("");
if (!success) revert TransferFailed();
}
```

RECOMMENDATION

Use a pull-based refund pattern. Store excess ETH in a pendingRefunds mapping and let callers claim it via a separate claimRefund() function. Alternatively, require exact payment with require(msg.value == bridgeCost).

6.2.19 setParams() Does Not Accrue Interest Before Rate Change

LOW FIXED

DESCRIPTION

When rate parameters are updated via InterestRateModel.setParams() or the model is swapped via BorrowModule.setInterestRateModel(), there is no mechanism to accrue pending interest under the old rates before the new rates take effect. The next call to _accrueGlobalInterest() applies the new rates retroactively to the entire elapsed period since lastGlobalAccrualTime, including time that passed before the rate change.

A rate increase causes borrowers to be overcharged for the pre-change period. A rate decrease causes the protocol and suppliers to lose interest they should have earned.

```
function setParams(/* ... */) external onlyRole(RATE_ADMIN_ROLE) {
baseRateBps = _baseRateBps;
multiplierBps = _multiplierBps;
// No _accrueGlobalInterest() call
}
function setInterestRateModel(address _model) external onlyRole(BORROW_ADMIN_ROLE)
{
require(_model != address(0), "ZERO_ADDRESS");
interestRateModel = IInterestRateModel(_model);
}
uint256 elapsed = block.timestamp - lastGlobalAccrualTime;
interest = interestRateModel.calculateInterest(totalBorrows, totalBorrows,
totalSupply, elapsed);
```

RECOMMENDATION

Call _accrueGlobalInterest() inside setInterestRateModel() before swapping the model. Expose a public accrueGlobalInterest() function. Add a callback in InterestRateModel.setParams() that triggers accrual before overwriting parameters.

6.2.20 minSupplyRateRequired Is Set But Never Enforced

LOW ACKNOWLEDGED

DESCRIPTION

TheminSupplyRateRequiredstate variable is set in the constructor and can be updated by admin viasetMinSupplyRate(), but it is never actually checked anywhere in the contract. The_leverage()function, which is the main entry point for deploying capital into Morpho, does not validate whether the current supply rate meets this minimum threshold before depositing. This means the rate protection is entirely absent despite appearing to be configured.

CODE LOCATION

File:contracts/strategies/MorphoLoopStrategy.sol

contracts/strategies/MorphoLoopStrategy.sol

```
minSupplyRateRequired = _minSupplyRate;
function setMinSupplyRate(uint256 _rate) external onlyRole(DEFAULT_ADMIN_ROLE) {
minSupplyRateRequired = _rate;
}
function _leverage(uint256 targetLeverage) internal {
// No check: require(currentSupplyRate >= minSupplyRateRequired)
// Proceeds to supply and borrow regardless of rate
}
```

RECOMMENDATION

Add a supply rate check at the start of_leverage():

Or remove the variable entirely if rate checks are not intended, to avoid giving a false sense of security.

```
uint256 currentRate = morpho.supplyAPY(marketParams);
require(currentRate >= minSupplyRateRequired, "Supply rate below minimum");
```

6.2.21 withdraw() Deducts Full Principal Regardless of Actual Amount Freed

LOW ACKNOWLEDGED

DESCRIPTION

Thewithdraw()function computesprincipalToWithdrawbased on the requested amount, then calls_deleverage()to free the collateral. However,_deleverage()may free less than requested due to iteration limits or liquidity constraints. Despite this,withdraw()deducts the fullprincipalToWithdrawfromprincipalDeposited, creating a mismatch between the tracked principal and the actual funds in the strategy. Over multiple partial withdrawals,principalDepositedcan underflow or reach zero while significant capital remains deployed.

CODE LOCATION

File:contracts/strategies/MorphoLoopStrategy.sol, Lines 308-333

contracts/strategies/MorphoLoopStrategy.sol (Lines 308-333)

```
function withdraw(uint256 amount) external override onlyRole(TREASURY_ROLE)
returns (uint256) {
uint256 totalVal = totalValue();
uint256 principalToWithdraw = (amount * principalDeposited) / totalVal;
_deleverage(targetBorrowAfter);
uint256 available = usdc.balanceOf(address(this));
uint256 actualWithdraw = available < amount ? available : amount;
principalDeposited -= principalToWithdraw; // Deducts full amount even if partial
usdc.safeTransfer(msg.sender, actualWithdraw);
```

```
return actualWithdraw;
}
```

RECOMMENDATION

Scale the principal deduction by the ratio of actual to requested withdrawal:

```
uint256 adjustedPrincipal = (principalToWithdraw * actualWithdraw) / amount;
principalDeposited -= adjustedPrincipal;
```

6.2.22 No Oracle Cardinality Check Causes TWAP Query to Revert

LOW FIXED

DESCRIPTION

The_getMarketInfo()function callsIPendleOracle(PENDLE_ORACLE).getPtToSyRate(market, TWAP_DURATION)without first checking whether the market's oracle has sufficient observation cardinality to support the requested TWAP duration. By default, all new Pendle markets haveobservationCardinality = 1, which is insufficient for a 900-second TWAP (requires cardinality of 85).

The oracle call reverts with "OLD" when cardinality is insufficient. Since there is notry/catcharound this call, the revert propagates up through_getMarketInfo()togetValidMarkets()andselectBestMarket(). A single uninitialized market in the whitelist causes the entire market selection to revert, blocking all other valid markets as well.

The contract readsobservationCardinalityfrom_storage()at Line 318 but discards it without any validation.

CODE LOCATION

File:contracts/PendleMarketSelector.sol, Line 335

contracts/PendleMarketSelector.sol (Lines 335-335)

```
function _getMarketInfo(address market) internal view returns (MarketInfo memory
info) {
// ...
(, , , , uint16 observationCardinality,) = IPendleMarket(market)._storage();
// observationCardinality is read but never checked
uint256 ptToSyRate = IPendleOracle(PENDLE_ORACLE).getPtToSyRate(market,
TWAP_DURATION);
// Reverts with "OLD" if cardinality insufficient, no try/catch
}
```

RECOMMENDATION

Wrap the oracle call intry/catchto gracefully skip uninitialized markets:

Also validate oracle readiness during whitelisting:

```
try IPendleOracle(PENDLE_ORACLE).getPtToSyRate(market, TWAP_DURATION) returns
(uint256 rate) {
ptToSyRate = rate;
} catch {
continue; // Skip this market
}
function whitelistMarket(address market, string calldata category) external
onlyRole(MARKET_ADMIN_ROLE) {
(bool increaseNeeded, , bool satisfied) =
IPendleOracle(PENDLE_ORACLE).getOracleState(market, TWAP_DURATION);
require(!increaseNeeded && satisfied, "ORACLE_NOT_READY");
// ... rest of whitelisting
}
```

6.2.23 emergencyWithdraw Emits Wrong ptBalance Value in Event

LOW ACKNOWLEDGED

DESCRIPTION

TheEmergencyWithdrawevent always emitsptRedeemed = 0becauseptBalanceis read after_redeemPt()has already set it to zero. Inside_redeemPt(),ptBalance -= ptAmountruns before the event emission inemergencyWithdraw(). So by the timeemit EmergencyWithdraw(ptBalance, balance)executes,ptBalanceis always zero regardless of how much PT was actually redeemed.

This causes off-chain monitoring, dashboards, and treasury accounting to record zero PT redeemed during emergency withdrawals, making post-incident forensics unreliable.

CODE LOCATION

File:contracts/strategies/PendleStrategyV2.sol, Lines 787-798

contracts/strategies/PendleStrategyV2.sol (Lines 787-798)

```
function emergencyWithdraw() external onlyRole(GUARDIAN_ROLE) {
_pause();
uint256 usdcOut = 0;
if (ptBalance > 0) {
usdcOut = _redeemPt(ptBalance); // Sets ptBalance to 0
}
uint256 balance = usdc.balanceOf(address(this));
emit EmergencyWithdraw(ptBalance, balance); // ptBalance is now 0
```

```
}
ptBalance -= ptAmount; // When ptAmount == ptBalance, this sets ptBalance = 0
```

RECOMMENDATION

CacheptBalancebefore the mutation:

```
function emergencyWithdraw() external onlyRole(GUARDIAN_ROLE) {
_pause();
uint256 ptToRedeem = ptBalance;
uint256 usdcOut = 0;
if (ptToRedeem > 0) {
usdcOut = _redeemPt(ptToRedeem);
}
uint256 balance = usdc.balanceOf(address(this));
emit EmergencyWithdraw(ptToRedeem, balance);
}
```

6.2.24 resetLastKnownPrice() Skips Staleness and Circuit Breaker Checks

LOW FIXED

DESCRIPTION

TheresetLastKnownPrice()function allows an oracle admin to setlastKnownPrice[token]directly from the latest Chainlink round data without checking staleness, round completeness, or the circuit breaker deviation threshold. The only validation isprice > 0. This means an admin can anchor the circuit breaker to a stale or manipulated price, which then affects all subsequentgetPrice()calls that compare against this anchor.

If the Chainlink feed is stale or returning an outdated price at the time of the reset, the anchor becomes incorrect. Future legitimate prices that deviate significantly from this bad anchor will trigger the circuit breaker unnecessarily, causing a denial of service for all protocol operations that depend on the oracle.

CODE LOCATION

File:contracts/PriceOracle.sol, Lines 72-79

contracts/PriceOracle.sol (Lines 72-79)

```
function resetLastKnownPrice(address token) external onlyRole(ORACLE_ADMIN_ROLE) {
FeedConfig memory config = feeds[token];
require(config.feed != address(0), "FEED_NOT_SET");
(, int256 answer, , ,) = AggregatorV3Interface(config.feed).latestRoundData();
require(answer > 0, "INVALID_PRICE");
lastKnownPrice[token] = uint256(answer);
```

```
// Missing: staleness check, roundId check, circuit breaker check
}
```

RECOMMENDATION

Apply the same staleness, round completeness, and deviation checks used ingetPrice():

```
function resetLastKnownPrice(address token) external onlyRole(ORACLE_ADMIN_ROLE) {
FeedConfig memory config = feeds[token];
require(config.feed != address(0), "FEED_NOT_SET");
(uint80 roundId, int256 answer, , uint256 updatedAt, uint80 answeredInRound) =
AggregatorV3Interface(config.feed).latestRoundData();
require(answer > 0, "INVALID_PRICE");
require(updatedAt >= block.timestamp - config.heartbeat, "STALE_PRICE");
require(answeredInRound >= roundId, "INCOMPLETE_ROUND");
lastKnownPrice[token] = uint256(answer);
}
```

6.2.25 Legacy withdraw() Does Not Update lastRecordedValue, Causing Fee Drift

LOW FIXED

DESCRIPTION

Neitherwithdraw()norwithdrawToVault()updatelastRecordedValueafter the withdrawal. When these functions are called withinMIN_ACCRUAL_INTERVAL(1 hour),_accrueFees()returns early without updating the baseline. The withdrawal then reducestotalValue()butlastRecordedValuestays at the stale pre-withdrawal level.

On the next accrual (when the interval passes),currentValue < lastRecordedValue, so no fees are accrued even though real yield was earned. For example, if 5,000 USDC of yield accrues and then a 50,000 USDC withdrawal happens within the interval, the accrual sees965,001 < 1,010,001and records zero fees. The protocol loses 2,000 USDC (40% fee on 5,000) per occurrence.

CODE LOCATION

File:contracts/TreasuryV2.sol, Lines 455-468 (withdraw()) and Lines 370-408 (withdrawToVault())

contracts/TreasuryV2.sol (Lines 455-468)

```
function withdraw(address to, uint256 amount) external nonReentrant whenNotPaused
onlyRole(VAULT_ROLE) {
if (amount == 0) revert ZeroAmount();
_accrueFees();
// ... withdrawal logic ...
asset.safeTransfer(to, amount);
```

```
emit Withdrawn(to, amount);
// Missing: lastRecordedValue = totalValue();
}
if (block.timestamp < lastFeeAccrual + MIN_ACCRUAL_INTERVAL) {
return; // No lastRecordedValue update
}
```

RECOMMENDATION

AddlastRecordedValue = totalValue()at the end of both withdrawal functions:

```
function withdraw(address to, uint256 amount) external nonReentrant whenNotPaused
onlyRole(VAULT_ROLE) {
if (amount == 0) revert ZeroAmount();
_accrueFees();
asset.safeTransfer(to, amount);
lastRecordedValue = totalValue();
emit Withdrawn(to, amount);
}
```

7. Executive Summary

Two independent softstack experts performed an unbiased and isolated audit of the smart contract provided by the Minted team. The main objective of the audit was to verify the security and functionality claims of the smart contract. The audit process involved a thorough manual code review and automated security testing.

Overall, the audit identified a total of 25 issues, classified as follows:

- No critical issues were found
- 4 high severity issues were found
- 9 medium severity issues were found
- 12 low severity issues were found
- No informational issues were found

The Minted team has successfully addressed all identified issues from the audit. All vulnerabilities have been mitigated based on the recommendations provided in the report. A follow-up review confirms that the fixes have been implemented effectively, ensuring the security and functionality of the smart contract. 5 findings were acknowledged by the team.

The audited codebase comprises 4,424 normalized source lines of code across 17 files.

8. About the Auditor

Established in 2017 under the name Chainsulting, and rebranded as softstack GmbH in 2023, softstack has been a trusted name in Web3 Security space. Within the rapidly growing Web3 industry, softstack provides a comprehensive range of offerings that include software development, cybersecurity, and consulting services. Softstack's competency extends across the security landscape of prominent blockchains like Solana, Tezos, TON, Ethereum and Polygon. The company is widely recognized for conducting thorough code audits aimed at mitigating risk and promoting transparency.

The firm's proficiency lies particularly in assessing and fortifying smart contracts of leading DeFi projects, a testament to their commitment to maintaining the integrity of these innovative financial platforms. To date, softstack plays a crucial role in safeguarding over $100 billion worth of user funds in various DeFi protocols.

Underpinned by a team of industry veterans possessing robust technical knowledge in the Web3 domain, softstack offers industry-leading smart contract audit services. Committed to evolving with their clients' ever-changing business needs, softstack's approach is as dynamic and innovative as the industry it serves.

Check our website for further information: https://softstack.io

9. Glossary

Term	Definition
CVSS	Common Vulnerability Scoring System
DeFi	Decentralized Finance
ERC	Ethereum Request for Comments
EVM	Ethereum Virtual Machine
LTV	Loan-to-Value ratio
TVL	Total Value Locked
TWAP	Time-Weighted Average Price