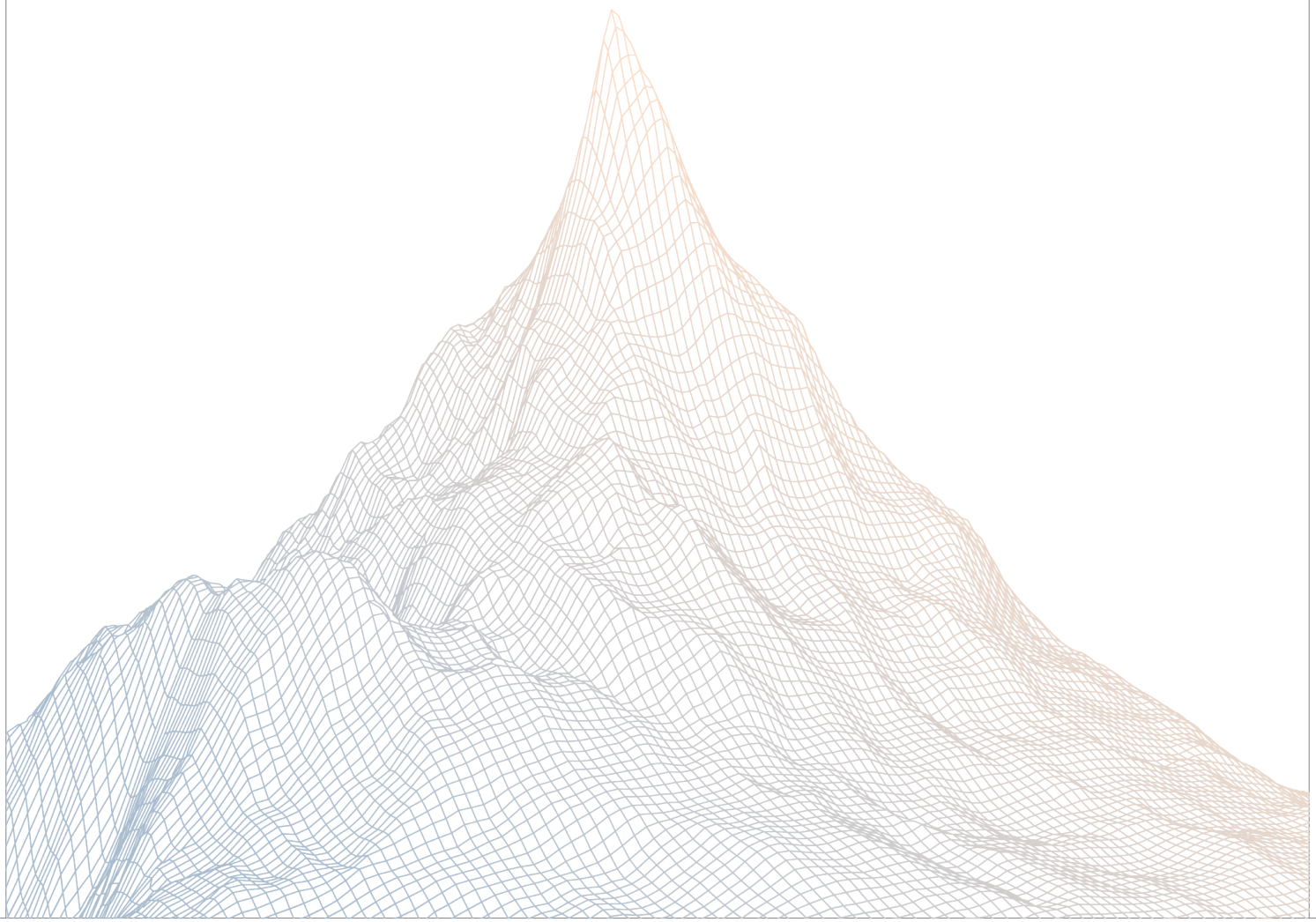
Zenith

UNIT

Smart Contract Security Assessment

VERSION 1.1



AUDIT DATES: August 24th to 25th, 2026

AUDITED BY: Dravee
Mario Poneder

Contents

1

Introduction

1.1 About Zenith

Zenith assembles auditors with proven track records: finding critical vulnerabilities in public audit competitions.

Our audits are carried out by a curated team of the industry's top-performing security researchers, selected for your specific codebase, security needs, and budget.

Learn more about us at https://zenith.security.

1.2 Disclaimer

This report reflects an analysis conducted within a defined scope and time frame, based on provided materials and documentation. It does not encompass all possible vulnerabilities and should not be considered exhaustive.

The review and accompanying report are presented on an "as-is" and "as-available" basis, without any express or implied warranties.

Furthermore, this report neither endorses any specific project or team nor assures the complete security of the project.

1.3 Risk Classification

SEVERITY LEVEL	IMPACT: HIGH	IMPACT: MEDIUM	IMPACT: LOW
Likelihood: High	Critical	High	Medium
Likelihood: Medium	High	Medium	Low
Likelihood: Low	Medium	Low	Low

2

Executive Summary

2.1 About UNIT

UNIT is a decentralized finance (DeFi) protocol designed to make stablecoin liquidity productive. Users can deposit supported stablecoins to mint UNIT, a stable asset backed 1:1 by the underlying stablecoin reserves. Rather than leaving the assets backing UNIT idle, the protocol deploys this liquidity into yield-generating strategies. Users can choose to stake their UNIT to participate in the rewards generated by the protocol while maintaining exposure to a stable, on-chain asset. By combining a 1:1-backed stable asset with an integrated staking mechanism, UNIT provides a simple way to put stablecoin liquidity to work while keeping the experience accessible and fully on-chain.

2.2 Scope

The engagement involved a review of the following targets:

Target	unit-core
Repository	https://github.com/AltitudeDP/unit-core
Commit Hash	`4a5a3b53c79b9bcf76350604d431838d9baaebcc`
Files	`src/**/*.sol`

Target	Mitigation Review
Repository	https://github.com/AltitudeDP/unit-core
Commit Hash	`fcbbc1e88ca3a57e8606e5fd426b12018f32ff99`
Files	`src/**/*.sol`

2.3 Audit Timeline

August 24, 2026	Audit start
August 25, 2026	Audit end
August 27, 2026	Report published

2.4 Issues Found

SEVERITY	COUNT
Critical Risk	0
High Risk	1
Medium Risk	3
Low Risk	7
Informational	6
Total Issues	**17**

3

Findings Summary

ID	Description	Status
H-1	Two-token JustLend reward claims revert because Minter2 cannot receive the TRX-vault callback	Resolved
M-1	Account-specific confiscation can be bypassed through transfers and staking	Resolved
M-2	Minting and redemption lack minimum-output protection	Resolved
M-3	Unlimited approvals allow compromised external dependencies to drain Minter2 balances	Resolved
L-1	Immediate USDT redemption depends on shared external liquidity	Acknowledged
L-2	Privileged backing removal makes fixed one-to-one accounting transfer deficits to later depositors	Resolved
L-3	Redemptions can consume undistributed liquid USDD rewards	Resolved
L-4	UNIT can be issued without adequate jUSDD share credit	Resolved
L-5	Sequential nonces can strand later permits and block redemptions	Acknowledged
L-6	Cumulative reward history is not enforced across Merkle root updates	Acknowledged
L-7	Ownership can be permanently renounced despite requiring future root updates	Resolved
I-1	Incorrect production addresses and unchecked dependency wiring permit zero-output mints	Resolved
I-2	MINTER_ROLE unnecessarily includes arbitrary third-party burning	Resolved
I-3	Signer reauthorization can reactivate outstanding permits	Acknowledged

ID	Description	Status
I-4	StakedUnit blocks share transfers but retains delegated ERC-4626 exits	Acknowledged
I-5	Small mints can consume USDT while issuing zero UNIT	Resolved
I-6	executeCall() can report success when status-code-based external operations failed	Resolved

4

Findings

4.1 High Risk

A total of 1 high risk findings were identified.

[H-1] Two-token JustLend reward claims revert because `Minter2` cannot receive the `TRX`-vault callback

SEVERITY: High	IMPACT: Medium
STATUS: Resolved	LIKELIHOOD: High

Target:

- src/Minter2.sol#L24-L45
- src/Minter2.sol#L145-L162

Description:

`Minter2` forwards proof-bound multi-token claims to the configured JustLend distributor:

```
function multiClaimJustLendRewards(
 IMultiMerkleDistributor.ClaimParam[] calldata claims
) external onlyRole(KEEPER_ROLE) {
 IMultiMerkleDistributor(
 JUSTLEND_DISTRIBUTOR
 ).multiClaim(claims);
}
```

However, `Minter2` implements neither a payable `receive()` nor a payable `fallback()`.

The deployed distributor iterates over every element of each `amounts` array and unconditionally calls its corresponding reward vault. Vault index one is `LendSafeVaultTRX` (`TQj9Za4HBdskm5PWPnrou1zb2FJpNCyyAo`), whose deployed bytecode performs logic equivalent to:

```
(bool success,) =
 receiver.call{value: tokenAmount}("");
require(success);
```

This call is performed even when `tokenAmount` is zero. Because it contains empty calldata,

it invokes the receiver's `receive()` or `fallback()`. `Minter2` has neither and therefore rejects the callback, reverting the complete claim atomically.

At review time, the JustLend API account documented in the README returned outstanding claims with two-element arrays of the form:

```
[USDD amount, 0 TRX]
```

This demonstrates the production claim format but does not assume that the current repository revision is already deployed at that account. The keeper cannot omit the zero-valued TRX element because the complete `amounts` array is included in the Merkle leaf.

Consequently, if `Minter2` is deployed as written, both the USDD and TRX portions of two-token claims are unharvestable. The rewards remain unclaimed rather than being transferred to an attacker, and principal collateral is not directly affected. Calling the distributor through `executeCall` does not bypass the issue because the distributor still sees `Minter2` as the claimant and sends the callback to it.

The existing claim test also does not detect the incompatibility because its etched distributor mock transfers only `amounts[0]` and does not reproduce the deployed per-token vault callbacks.

Recommendations:

It is recommended to add an intentional payable receiver with no external calls or sensitive state changes:

```
event NativeValueReceived(
 address indexed sender,
 uint256 amount
);

receive() external payable {
 emit NativeValueReceived(msg.sender, msg.value);
}
```

An unrestricted receiver avoids introducing another immutable external dependency. If sender filtering is required, validate the immediate TRX reward vault rather than `JUSTLEND_DISTRIBUTOR`, and make the allowed vault configurable to support rotation.

The protocol should also define how positive TRX rewards are handled. They can be recovered through the existing admin-only `executeCall`, but a dedicated conversion or distribution path may provide clearer accounting and operational controls.

Replace or supplement the current mock with one that reproduces the distributor's

per-token vault calls. Test two-element claims with both zero and positive TRX amounts using a TRON-native integration environment or an accurate production-bytecode simulation.

UNIT: Resolved with `@e684052b8a ...` - by adding a payable receive handler for JustLend TRX reward callbacks.

Zenith: Verified.

4.2 Medium Risk

A total of 3 medium risk findings were identified.

[M-1] Account-specific confiscation can be bypassed through transfers and staking

SEVERITY: Medium	IMPACT: Medium
STATUS: Resolved	LIKELIHOOD: Medium

Target:

- src/Unit.sol#L31-L34
- src/StakedUnit.sol#L8-L26

Description:

`Unit.confiscate` transfers only the specified amount from the target's balance at execution. It records no persistent restriction that prevents the target from transferring existing or subsequently received UNIT.

```
function confiscate(address from, address to, uint256 value)
 external
 onlyRole(DEFAULT_ADMIN_ROLE)
{
 _transfer(from, to, value);
 emit Confiscated(from, to, value);
}
```

A target observing a pending confiscation can transfer enough UNIT that the requested amount exceeds their remaining balance, causing the transaction to revert. The target can also deposit UNIT into `StakedUnit`, where the underlying becomes pooled vault custody and the target receives an equivalent sUNIT balance.

Although sUNIT transfers are prohibited, minting and burning remain permitted:

```
function _update(address from, address to, uint256 value)
    internal override {
 if (from ≠ address(0) && to ≠ address(0)) {
```

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

Consequently, the target can call the inherited ERC-4626 `redeem` function as the share owner, burn their shares without an allowance, and send the underlying UNIT to any receiver. `StakedUnit` provides no privileged operation for freezing or confiscating a specific account's shares.

Confiscating or burning UNIT held by the vault is not an account-specific remedy because the target's shares remain outstanding. It instead creates a pooled deficit under the fixed 1:1 accounting, allowing early redeemers to exit fully while later redeemers bear the shortfall.

The account can also receive and spend new UNIT after a completed confiscation. Therefore, the documented regulatory freeze capability does not persistently restrict liquid balances and does not cover staked positions.

Recommendations:

It is recommended to first define whether `confiscate` is intended to provide persistent account restrictions, one-time seizure, or both.

If persistent account control is required, use a single authoritative restriction registry. Enforce restrictions against senders and receivers in UNIT transfers, with an explicit authorized bypass for confiscation. `StakedUnit` should additionally validate the caller and receiver during deposits or mints, and the caller, owner, and receiver during withdrawals or redemptions.

Add a privileged vault operation that atomically burns the target's shares and transfers an equal amount of available backing to the designated recipient. It must not remove backing without cancelling the corresponding shares, and its behavior when the vault is undercollateralized must be defined.

Persistent restrictions prevent activity after inclusion but cannot independently prevent a target from front-running a publicly visible restriction transaction. Private or priority submission should be used where available, or this residual ordering limitation should be documented.

If staked positions are intentionally outside the confiscation perimeter, document that limitation and avoid describing `confiscate` as a freeze.

UNIT: Resolved with `@bfe98587e2 ...` - by adding persistent account freezing, blocking frozen vault operations, and safely confiscating staked positions.

Zenith: Verified.

[M-2] Minting and redemption lack minimum-output protection

SEVERITY: Medium	IMPACT: Medium
STATUS: Resolved	LIKELIHOOD: Low

Target:

- src/Minter2.sol#L96-L131
- src/Minter2.sol#L164-L185

Description:

`Minter2.mint` and `Minter2.redeem` commit the caller to a fixed input without allowing them to enforce a minimum output. The authorized EIP-712 signer approves the account, input, operation mode, nonce, and deadline, while output remains dependent on mutable PSM fees at execution.

The user directly submits the transaction and can inspect current fees, but cannot encode an acceptable tolerance against a fee change between submission and inclusion.

Minting

Minting first transfers the caller's USDT and executes the PSM swap. UNIT output is derived afterward from the received USDD:

```
USDT.safeTransferFrom(msg.sender, address(this), assets);

uint256 usddBefore = USDD.balanceOf(address(this));
PSM.sellGem(address(this), assets);
uint256 usddReceived =
 USDD.balanceOf(address(this)) - usddBefore;

if (
 usddReceived > 0
 && jUSDD.mint(usddReceived) ≠ 0
) revert OperationFailed();

return usddReceived / 1e12;
```

While PSM selling remains enabled, `tin` is applied in WAD units:

```
USDD output = assets × 1e12 × (1e18 − tin) / 1e18
```

At `tin = 1e18`, the complete USDT input is charged as a fee and zero USDD is sent to Minter2. Positive USDD output below `1e12` wei is deposited into jUSDD but floors to zero UNIT.

The public function explicitly accepts this zero result:

```
uint256 unitToMint = _mintInternal(assets);
if (unitToMint > 0) {
 if (stake) {
 UNIT.mint(address(this), unitToMint);
 stakedUnit.deposit(unitToMint, msg.sender);
 } else {
 UNIT.mint(msg.sender, unitToMint);
 }
}
emit Minted(msg.sender, unitToMint);
```

The transaction can therefore succeed, consume the permit nonce, and emit `Minted(account, 0)` after taking the caller's complete USDT input.

Redemption

Redemption burns a fixed amount of UNIT before calculating USDT output:

```
if (unstake) {
 stakedUnit.withdraw(
 assets,
 address(this),
 msg.sender
 );
 UNIT.burn(address(this), assets);
} else {
 UNIT.burn(msg.sender, assets);
}

_redeemInternal(assets);
```

While PSM buying remains enabled, output is calculated from the execution-time `tout` value:

```
uint256 tout = PSM.tout();
```

```
uint256 gemAmt =
 (assets * 1e18) / (1e18 + tout);

uint256 usddRequired =
 gemAmt * 1e12
 + (gemAmt * tout) / 1e6;

// ...

PSM.buyGem(address(this), gemAmt);
USDT.safeTransferFrom(
 address(this),
 msg.sender,
 gemAmt
);
```

Any positive `tout` makes a redemption of one base unit of UNIT floor to zero USDT. For a material redemption, an extreme `tout` is required to produce zero output. When `gemAmt` is zero, the zero-amount PSM exchange and final transfer can complete successfully, leaving the preceding UNIT burn intact.

Failed external calls revert all preceding state atomically, but successful zero-output or poor-output settlements do not.

At the time of review, the live PSM reported `tin = 0` and `tout = 0`. Changing either value requires authorized PSM governance, so these are not permissionless attacks. However, the scoping documentation states that integrated-protocol administrator values have no assumed limitations.

Recommendations:

It is recommended to add caller-provided `minUnitOut` and `minUsdtOut` parameters and enforce them against actual output. Positive input should never produce zero output, regardless of the selected minimum.

For `stake == true`, `minUnitOut` should represent the minimum UNIT-denominated sUNIT shares expected, currently equivalent under the vault's fixed 1:1 conversion.

```
error InsufficientOutput();

function mint(
 uint256 assets,
 bool stake,
 uint256 minUnitOut,
 uint256 deadline,
 bytes calldata signature
```

```
) external {
 // Perform existing permit validation.

 uint256 unitToMint = _mintInternal(assets);

 if (
 (assets > 0 && unitToMint == 0)
 || unitToMint < minUnitOut
 ) revert InsufficientOutput();

 // Issue liquid UNIT or 1:1 sUNIT shares.
}

function redeem(
 uint256 assets,
 bool unstake,
 uint256 minUsdtOut,
 uint256 deadline,
 bytes calldata signature
) external {
 // Perform existing permit validation.

 // Optionally calculate and reject nominal zero output
 // before burning to avoid unnecessary external calls.

 // Burn liquid UNIT or withdraw and burn staked UNIT.

 uint256 balanceBefore =
 USDT.balanceOf(msg.sender);

 _redeemInternal(assets);

 uint256 actualOut =
 USDT.balanceOf(msg.sender) - balanceBefore;

 if (
 (assets > 0 && actualOut == 0)
 || actualOut < minUsdtOut
 ) revert InsufficientOutput();
}
```

Reverting after either internal operation atomically restores all preceding transfers, swaps, deposits, and burns. Measuring the recipient's actual USDT balance increase also protects against a possible transfer fee.

Because the user authenticates the transaction calldata, the minimum-output parameters do not need to be included in the EIP-712 type hashes to protect the user. Include them in

the signed permits only if the authorized signer must approve the economic quote. Short permit deadlines should be additional protection, not a replacement for output bounds.

UNIT: Resolved with `@9aedc0252a ...` - by adding caller-enforced minimum outputs and rejecting zero-output mint and redemption operations.

Zenith: Verified.

[M-3] Unlimited approvals allow compromised external dependencies to drain Minter2 balances

SEVERITY: Medium	IMPACT: Medium
STATUS: Resolved	LIKELIHOOD: Low

Target:

- Minter2.sol#constructor

Description:

Minter2 permanently grants `type(uint256).max` allowances to several contracts:

Allowance	Threat model
`USDT → Minter2`	No additional external trust; Minter2 is its own spender.
`USDT → PSM.gemJoin()`	Required for `sellGem()`. No permissionless path was identified allowing another caller to spend Minter2's USDT.
`USDD → PSM`	Required for `buyGem()`. Normal calls spend the caller's USDD, not an arbitrary account's.
`USDD → jUSDD`	**Critical trust boundary:** jUSDD is a `CErc20Delegator` whose implementation is replaceable through governance.
`UNIT → StakedUnit`	No permissionless path was identified allowing a user to make the vault pull assets from Minter2.

The jUSDD allowance creates the strongest additional exposure. JustLend's jToken markets are upgradeable proxy/delegator contracts, and the jUSDD implementation can be replaced while retaining the same spender address. (JustLend Documentation)

Consequently, a compromised future implementation executing through the jUSDD delegator could directly call:

```
USDD.transferFrom(address(minter2), attacker, amount); // <--- with
    Minter2's whole balance
```

The USDD token sees the already-approved jUSDD proxy as `msg.sender`, so no further permission from UNIT is required. The attacker can pull up to Minter2's entire liquid USDD balance.

This makes UNIT security dependent not only on the jUSDD functionality it currently integrates with, but also on **every future JustLend implementation upgrade for that address**.

The same general assumption applies to other permanently approved external spenders, although no equivalent mutable implementation path was identified for the in-scope StakedUnit or normal PSM interaction.

Risk Analysis

Impact is High because a compromised approved spender can bypass Minter2's access controls entirely and drain the corresponding token balance directly through `transferFrom`. The loss is bounded only by the balance Minter2 holds at that time.

Likelihood is Low because exploitation requires failure of an external trusted dependency rather than a permissionless Minter2 exploit.

Recommendations:

Prefer **operation-scoped allowances** for external dependencies:

```
USDD.forceApprove(address(jUSDD), usddAmount);
jUSDD.mint(usddAmount);
USDD.forceApprove(address(jUSDD), 0);
```

The same pattern can be applied where practical to PSM/gemJoin and StakedUnit interactions.

This is the strongest protection because, outside the short execution window, a dependency has **zero standing authority** over Minter2's assets. The tradeoff is additional TRON Energy consumption and token calls; it is operationally less convenient than a permanent approval, but materially reduces the trust boundary.

An alternative protection with a pause **would be insufficient**: an already-approved malicious jUSDD can call `transferFrom()` even while Minter2's own functions are paused. The allowance itself must be revoked to terminate that authority.

UNIT: Resolved with `@d0b93a8eca ...`

- Removed permanent type(uint256).max USDD allowances from Minter2 constructor.
- Switched to operation-scoped approvals in _depositToJustLend() and _redeemInternal(), granting exact allowances only for the duration of the external call and resetting them to 0 immediately afterward.

Zenith: Verified.

4.3 Low Risk

A total of 7 low risk findings were identified.

[L-1] Immediate USDT redemption depends on shared external liquidity

SEVERITY: Low	IMPACT: Low
STATUS: Acknowledged	LIKELIHOOD: Low

Target:

- src/Minter2.sol#L164-L185

Description:

The protocol intentionally supplies every positive USDD mint output to the shared JustLend jUSDD market:

```
uint256 usddReceived =
 USDD.balanceOf(address(this)) - usddBefore;

if (
 usddReceived > 0
 && jUSDD.mint(usddReceived) ≠ 0
) revert OperationFailed();
```

Rewards or donations may leave incidental loose USDD in Minter2, but the protocol maintains no configured minimum liquid reserve.

During redemption, Minter2 first uses any loose USDD and requests the shortfall from jUSDD. It then exchanges USDD for USDT through the PSM:

```
uint256 usddBalance =
 USDD.balanceOf(address(this));

if (
 usddRequired > usddBalance
 && jUSDD.redeemUnderlying(
```

```
 usddRequired - usddBalance
 ) != 0
) revert OperationFailed();

PSM.buyGem(address(this), gemAmt);
USDT.safeTransferFrom(
 address(this),
 msg.sender,
 gemAmt
);
```

Immediate redemption therefore has two independent external-liquidity requirements:

1. **JustLend liquidity:** Minter2 must recover sufficient USDD from its jUSDD position. High market utilization can leave the market with insufficient cash even when Minter2's claim remains nominally solvent.
2. **PSM liquidity:** The PSM and GemJoin must have sufficient USDT inventory and buying must remain enabled.

Earlier redeemers, other JustLend borrowers, or other PSM users can consume available liquidity before later UNIT holders redeem. Draining either venue requires substantial capital, and no capital-efficient permissionless depletion strategy has been established.

External-call failures revert atomically, including the preceding UNIT burn. Holders retain their UNIT but may be unable to convert it into USDT until liquidity returns.

Actual JustLend bad debt, malicious implementation replacement, or USDD insolvency are separate external trust risks rather than proof of this temporary-liquidity condition. They can nevertheless make the loss of redemption availability permanent or reduce the economic value of backing.

The dependency follows the documented architecture of supplying USDD to JustLend and settling through the PSM. This is therefore an operational-resilience limitation rather than an undisclosed accounting defect.

Recommendations:

It is recommended to explicitly document that immediate USDT redemption is not guaranteed when JustLend or PSM liquidity is unavailable.

If stronger availability guarantees are required:

- retain a risk-sized liquid USDD reserve to mitigate JustLend cash shortages;
- retain or arrange separate USDT liquidity for PSM outages;
- cap external exposure relative to observable market liquidity;

- monitor Minter2's jUSDD claim, JustLend cash, PSM buying status, and GemJoin USDT inventory;
- provide queued or delayed redemption during temporary illiquidity;
- optionally allow users to accept direct USDD settlement when the PSM is unavailable.

Any `maxRedeemable` value should be presented as a non-binding estimate based on:

- loose USDD;
- the lesser of Minter2's jUSDD claim and current JustLend cash;
- PSM USDT inventory adjusted for `tout`.

A redemption queue should not permanently burn UNIT before assets become available unless the holder receives an enforceable redemption claim. Partial or emergency settlement must burn UNIT only in proportion to assets delivered and enforce caller-defined minimum output.

UNIT: Acknowledged.

- This is an accepted operational model inherent to JustLend and PSM integrations.
- If external liquidity in JustLend or PSM is temporarily depleted, redeem() reverts atomically without burning user UNIT.
- Documented external liquidity dependencies and operational monitoring requirements in protocol documentation.

[L-2] Privileged backing removal makes fixed one-to-one accounting transfer deficits to later depositors

SEVERITY: Low	IMPACT: Low
STATUS: Resolved	LIKELIHOOD: Low

Target:

- src/StakedUnit.sol#L13-L18
- src/Unit.sol#L27-L34

Description:

`StakedUnit` intentionally uses fixed identity conversions independently of its actual UNIT balance and outstanding share supply:

```
function _convertToShares(
 uint256 assets,
 Math.Rounding
) internal pure override returns (uint256) {
 return assets;
}

function _convertToAssets(
 uint256 shares,
 Math.Rounding
) internal pure override returns (uint256) {
 return shares;
}
```

This preserves the documented one-UNIT-per-share rate while:

```
totalAssets() >= totalSupply()
```

Under the intended UNIT integration, an untrusted user cannot independently violate this condition. `StakedUnit` has no administrative withdrawal or external investment logic. A deficit requires privileged backing removal, principally:

```
function burn(
 address from,
 uint256 assets
) external onlyRole(MINTER_ROLE) {
 _burn(from, assets);
}

function confiscate(
 address from,
 address to,
 uint256 value
) external onlyRole(DEFAULT_ADMIN_ROLE) {
 _transfer(from, to, value);
}
```

If UNIT is burned or confiscated from `StakedUnit` without cancelling an equal number of shares:

- previews continue quoting one UNIT per share;
- inherited `maxRedeem` and `maxWithdraw` overstate executable liquidity;
- deposits and mints remain open at par;
- early holders can consume all available backing;
- later holders revert or absorb the complete deficit;
- new deposits can be captured by legacy underbacked shares.

For example, assume an existing holder owns 100 shares after backing falls to 50 UNIT. A new user deposits 50 UNIT and receives 50 shares. The vault then holds 100 UNIT against 150 shares. The existing holder can redeem 100 shares for all 100 UNIT, leaving the new user with 50 unbacked shares.

Once a privileged deficit exists, exploiting later deposits requires no additional privilege. Nevertheless, the requirement for trusted-role action materially reduces likelihood.

Recommendations:

It is recommended to prohibit burning or confiscating UNIT from `StakedUnit` unless an equal number of corresponding shares is atomically cancelled. If account-specific seizure is required, implement a privileged StakedUnit operation that burns the target's shares and transfers equal backing in the same transaction.

The protocol must also define one explicit deficit policy.

If permanent 1:1 redemption must be preserved, stop deposits, mints, withdrawals, and redemptions while underbacked and resume only after full recapitalization:

```
error VaultInsolvent();

function _isSolvent() internal view returns (bool) {
 return totalAssets() >= totalSupply();
}

function maxDeposit(
 address receiver
) public view override returns (uint256) {
 return _isSolvent()
 ? super.maxDeposit(receiver)
 : 0;
}

function maxMint(
 address receiver
) public view override returns (uint256) {
 return _isSolvent()
 ? super.maxMint(receiver)
 : 0;
}

function maxRedeem(
 address owner
) public view override returns (uint256) {
 return _isSolvent()
 ? super.maxRedeem(owner)
 : 0;
}
```

Inherited `maxWithdraw` derives from `maxRedeem`, so it also becomes zero while insolvent. Internal `_deposit` and `_withdraw` guards should additionally reject insolvency to prevent future internal callers from bypassing public limits.

Alternatively, if losses should be allocated pro rata, replace identity conversion with asset-to-supply-based accounting. This changes the documented fixed 1:1 product and reintroduces exchange-rate and donation considerations, so it should be adopted only as a deliberate product change rather than combined with the pause-and-recapitalize policy.

UNIT: Resolved with `@bde5ceea08 ...` - by disabling deposits, mints, withdrawals, and redemptions while the vault is undercollateralized.

Zenith: Verified.

[L-3] Redemptions can consume undistributed liquid USDD rewards

SEVERITY: Low	IMPACT: Low
STATUS: Resolved	LIKELIHOOD: Low

Target:

- src/Minter2.sol#L133-L150
- src/Minter2.sol#L175-L185

Description:

The documented reward workflow claims USDD into `Minter2` and subsequently calls `distributeRewards` to supply it to JustLend and mint an equivalent amount of UNIT to the reward distributor:

```
function distributeRewards(
 uint256 usddAmount,
 address distributor
) external onlyRole(KEEPER_ROLE) {
 _checkRole(DISTRIBUTOR_ROLE, distributor);
 if (usddAmount == 0) return;

 if (jUSDD.mint(usddAmount) ≠ 0) {
 revert OperationFailed();
 }

 UNIT.mint(distributor, usddAmount / 1e12);
}
```

Although the claimed USDD is economically earmarked for this workflow, `multiClaimJustLendRewards` neither measures the received balance delta nor records it as reserved inventory.

Consequently, `_redeemInternal` treats the complete liquid USDD balance as available redemption liquidity:

```
uint256 usddBalance =
 USDD.balanceOf(address(this));
```

```
if (
 usddRequired > usddBalance
 && jUSDD.redeemUnderlying(
 usddRequired - usddBalance
 ) ≠ 0
) {
 revert OperationFailed();
}

PSM.buyGem(address(this), gemAmt);
```

If an authorized redemption executes after a successful claim but before distribution, the PSM can consume the claimed USDD. Because the liquid balance was sufficient, the corresponding principal remains invested in jUSDD.

A subsequent distribution for the originally claimed amount then reverts during `jUSDD.mint`, either through its underlying transfer or its returned error code. Distributing only the remaining liquid balance instead underfunds the intended reward issuance.

The protocol remains solvent because the redemption burns an equivalent UNIT claim while leaving additional backing in the retained jUSDD position. The impact is delayed or underfunded reward distribution rather than theft or principal loss. Recovery requires realizing the retained jUSDD surplus or providing equivalent unreserved USDD. Additional earmarked rewards only shift the aggregate shortfall.

Recommendations:

It is recommended to claim and distribute rewards atomically using the measured USDD balance increase:

```
function claimAndDistributeRewards(
 IMultiMerkleDistributor.ClaimParam[]
 calldata claims,
 address distributor
) external onlyRole(KEEPER_ROLE) {
 _checkRole(DISTRIBUTOR_ROLE, distributor);

 uint256 balanceBefore =
 USDD.balanceOf(address(this));

 IMultiMerkleDistributor(
 JUSTLEND_DISTRIBUTOR
 ).multiClaim(claims);

 uint256 claimed =
```

```
  USDD.balanceOf(address(this))
  - balanceBefore;

  if (claimed == 0) return;
  if (jUSDD.mint(claimed) != 0) {
  revert OperationFailed();
  }

  UNIT.mint(distributor, claimed / 1e12);
 }
```

Remove the standalone claim path or route it through the atomic workflow so the unsafe sequence cannot remain in use.

If separate operations must be preserved, increment `pendingUsddRewards` by each measured claim delta, exclude that amount when calculating redemption liquidity, require distributions not to exceed it, and decrement it atomically with successful distribution.

UNIT: Resolved with `@477dcfa5f9 ...` - by replacing separate claim and distribution paths with an atomic balance-delta-based reward workflow.

Zenith: Verified.

[L-4] `UNIT` can be issued without adequate `jUSDD` share credit

SEVERITY: Low	IMPACT: Low
STATUS: Resolved	LIKELIHOOD: Low

Target:

- src/Minter2.sol#L133-L142
- src/Minter2.sol#L164-L172

Description:

Both jUSDD deposit paths treat a zero status code from `jUSDD.mint` as proof that the nominal USDD amount was adequately credited:

```
if (
 usddReceived > 0
 && jUSDD.mint(usddReceived) ≠ 0
) {
 revert OperationFailed();
}

return usddReceived / 1e12;
```

The reward path makes the same assumption:

```
if (jUSDD.mint(usddAmount) ≠ 0) {
 revert OperationFailed();
}

UNIT.mint(
 distributor,
 usddAmount / 1e12
);
```

JustLend calculates minted shares using truncating division:

```
minted shares =
 floor(actual underlying received
```

```
  × 1e18 / exchange rate)
```

Its successful status and post-mint verification hook do not require a positive number of shares. Consequently, success does not prove that Minter2 received sufficient economic credit to back the UNIT being issued.

If jUSDD supply falls to dust, an attacker controlling the residual shares can donate USDD directly to the market and inflate its exchange rate. For example, when the attacker owns one share and donated underlying `X` exceeds a subsequent Minter2 deposit `A`, that deposit can produce:

```
floor(A / X) = 0 shares
```

The jUSDD call still receives the USDD and returns success. Minter2 then issues UNIT from the nominal amount despite receiving no shares. The attacker can redeem the residual shares before Minter2 recovers the underlying, capturing both the donation and Minter2's deposit and leaving the new UNIT without corresponding backing.

Material undercredit can also occur before the result reaches zero: a deposit may receive a small number of shares whose post-mint value is substantially below the UNIT liability issued from the nominal amount.

Canonical USDD is not fee-on-transfer or rebasing, and ordinary jUSDD rounding is currently negligible. The live market also has substantial supply and liquidity, making the required exchange-rate manipulation economically impractical under present conditions. The issue becomes relevant if the market is wound down or its share supply otherwise approaches zero.

Recommendations:

It is recommended to use one checked jUSDD-deposit helper for both issuance paths. Measure the share delta, reject zero credit, value the received shares using the post-mint exchange rate, and reject any material discrepancy:

```
uint256 sharesBefore =
 IERC20(address(jUSDD)).balanceOf(
 address(this)
 );

if (jUSDD.mint(usddAmount) ≠ 0) {
 revert OperationFailed();
}

uint256 sharesReceived =
 IERC20(address(jUSDD)).balanceOf(
```

```
 address(this)
 ) - sharesBefore;

if (sharesReceived == 0) {
 revert InsufficientBackingCredit();
}

uint256 creditedBacking = Math.mulDiv(
 sharesReceived,
 jUSDD.exchangeRateStored(),
 1e18
);

uint256 creditedUnits =
 creditedBacking / 1e12;

if (
 nominalUnits > creditedUnits
 && nominalUnits - creditedUnits
 > MAX_UNIT_ROUNDING_LOSS
) {
 revert InsufficientBackingCredit();
}

uint256 unitToMint = Math.min(
 nominalUnits,
 creditedUnits
);
```

Set `MAX_UNIT_ROUNDING_LOSS` to a tightly bounded number of UNIT base units that covers only expected jUSDD rounding. Do not issue the nominal amount when credited value is lower unless an explicit rounding reserve covers the difference.

As an additional invariant, verify after each deposit that Minter2's liquid USDD plus the underlying value of all its jUSDD shares covers total UNIT supply normalized to 18 decimals.

Extend the jUSDD interface with `balanceOf` and `exchangeRateStored`, and use overflow-safe multiplication when valuing shares.

UNIT: Resolved with `@d271ba8a99 ...` and `@d9d33ef08d ...` - by rejecting zero-share deposits and capping UNIT issuance to post-mint jUSDD backing with tightly bounded rounding loss.

Zenith: Verified.

[L-5] Sequential nonces can strand later permits and block redemptions

SEVERITY: Low	IMPACT: Low
STATUS: Acknowledged	LIKELIHOOD: Low

Target:

- Minter2.sol#mint
- Minter2.sol#redeem

Description:

Mint and redeem share a strictly sequential per-user nonce. If the current permit expires or otherwise reverts, its nonce is not consumed, making already-issued higher-nonce permits unusable.

Affected users may therefore be unable to redeem UNIT until a valid operation consumes the missing nonce.

Risk Analysis

Impact is Medium because an affected user can temporarily lose access to redemption despite holding valid UNIT and a later signed redeem permit.

Likelihood is Low because recovery is possible by signing a new permit for the unchanged nonce, but this depends on the off-chain signer supporting nonce reuse. A backend that advances its own nonce when issuing rather than when observing successful on-chain consumption may require manual recovery.

Hypothesis / Severity Conditions

Severity is confirmed or strengthened if the signer:

- issues multiple outstanding permits per account;
- advances its nonce when signing rather than after successful execution;
- cannot automatically reissue an unconsumed nonce.

Severity decreases if only one permit can be outstanding and the signer always reads the on-chain nonce and freely reissues failed/expired permits.

Step-by-Step Example

1. Alice holds `500,000 UNIT`.
2. She receives redeem permits for nonces `20` and `21`.
3. The nonce-20 permit expires; the on-chain nonce remains `20`.
4. Alice submits the nonce-21 permit, but Minter2 expects nonce `20`, so it fails.
5. Alice cannot use the later permit until a nonce-20 operation succeeds.
6. If the backend has already advanced past nonce `20`, redemption requires backend nonce reconciliation or manual intervention.

Recommendations:

The signer should always derive its nonce from `Minter2.nonces(account)` and allow replacement signatures for unconsumed nonces.

If multiple simultaneous permits are required, consider unordered/bitmap nonces or a user-controlled nonce invalidation mechanism.

UNIT: Acknowledged.

- The backend signer generates single on-demand permits by fetching the latest on-chain nonce directly from Minter2.nonces(account).
- Pre-signing batches of sequential nonces is avoided by design.
- Expired or reverted permits leave the on-chain nonce intact, allowing the user to seamlessly request a fresh signature with the current nonce.

[L-6] Cumulative reward history is not enforced across Merkle root updates

SEVERITY: Low	IMPACT: Low
STATUS: Acknowledged	LIKELIHOOD: Low

Target:

- CumulativeMerkleDrop.sol#setMerkleRoot
- CumulativeMerkleDrop.sol#claim

Description:

`CumulativeMerkleDrop` stores only one active `merkleRoot`, while `cumulativeClaimed` only tracks amounts already claimed.

This creates an important asymmetry across root updates:

- if a new root increases an account's `cumulativeAmount`, the user can claim the delta;
- if a new root decreases it and the user already claimed the higher amount, nothing is clawed back;
- if the user had not yet claimed the higher amount, replacing the root invalidates the old proof and the lower entitlement becomes the only claimable one.

Therefore a mistaken or non-monotonic root can reduce or erase previously published but unclaimed rewards.

The current `expectedMerkleRoot` check only prevents claims against a stale root; it does not enforce monotonic cumulative entitlements across roots.

This may be intentional if root replacement is meant to revoke or correct unclaimed allocations. If roots are instead intended to represent cumulative, non-decreasing reward history, the off-chain generator must preserve that invariant because the contract cannot infer it from opaque Merkle roots.

Keeping old roots valid would protect users against accidental decreases, but introduces the opposite tradeoff: an accidentally over-allocated historical root remains exploitable during the period in which it stays authorized. Any multi-root design must still store and explicitly authorize every valid root; accepting caller-supplied roots without an on-chain authorization check would allow forged trees and arbitrary claims.

Recommendations:

- If cumulative entitlements are intended to be monotonic, enforce this invariant in the off-chain tree-generation process and add tests ensuring no account's cumulative amount decreases between roots.
- This is important: if a new root contains entitlements to new amounts instead of computing previous + new amounts, users won't get their full due.
- If previously published entitlements should remain claimable, consider storing multiple explicitly authorized roots instead of invalidating the previous root immediately (using `mapping(bytes32 ⇒ bool) validMerkleRoots;`).
- If erroneous roots must be revocable, keep the current single-root model or provide explicit root activation/revocation logic, accepting that unclaimed entitlements can change until the root is finalized.
- In all cases, only proofs against roots explicitly stored or authorized on-chain should be accepted.

UNIT: Acknowledged.

- Monotonic cumulative reward progression (cumulativeAmount(n) = cumulativeAmount(n-1) + delta) is strictly enforced in the off-chain Merkle tree generation pipeline.
- Pre-deployment scripts validate that no account's cumulative entitlement decreases between root updates.
- Retaining a single active merkleRoot allows the protocol owner to correct any erroneous distribution before claims are executed.

[L-7] Ownership can be permanently renounced despite requiring future root updates

SEVERITY: Low	IMPACT: Low
STATUS: Resolved	LIKELIHOOD: Low

Target:

- CumulativeMerkleDrop.sol — inherited `Ownable.renounceOwnership()`
- CumulativeMerkleDrop.sol#setMerkleRoot()

Description:

`CumulativeMerkleDrop` inherits `renounceOwnership()`, allowing the current owner to permanently remove the only authority capable of updating `merkleRoot`. The distributor is designed around recurring cumulative root updates, so ownership renouncement has no apparent useful protocol lifecycle and can permanently prevent future reward distributions.

Previously published claims remain executable, but new entitlements cannot be introduced after ownership is renounced. Recovery is impossible because ownership cannot subsequently be restored through the contract.

Recommendations:

Override `renounceOwnership()` to revert if an owner is permanently required:

```
function renounceOwnership() public override onlyOwner {
 revert();
}
```

Using `Ownable2Step` for transfers additionally reduces accidental ownership loss.

UNIT: Resolved with `@615de37fcd ...`

- Replaced Ownable with Ownable2Step in CumulativeMerkleDrop.sol for safe two-step ownership transfers.
- Overrode renounceOwnership() to revert with CannotRenounceOwnership, preventing accidental loss of the ability to update the Merkle root.

Zenith: Verified.

4.4 Informational

A total of 6 informational findings were identified.

[I-1] Incorrect production addresses and unchecked dependency wiring permit zero-output mints

SEVERITY: Informational	IMPACT: Informational
STATUS: Resolved	LIKELIHOOD: Low

Target:

- src/Minter2.sol#L65-L114
- src/Minter2.sol#L164-L172

Description:

The production addresses documented in the README contain multiple errors:

- The USDT Base58 address uses `JE` instead of `je` and has an invalid checksum.
- The USDD Base58 address `TPYmHEhy5n8TCEfYGqW2rPxsghSfzghPDn` resolves to legacy `USDDOLD` at `0x94f24e99...`, not the paired active USDD address `0xE91A7411....`
- The documented PSM Base58 address has an invalid checksum.
- The documented PSM hex address `0xB50Eb419...` is the USDT GemJoin. The active PSM is `TBXW4hS5...` / `0x1113ae08....`

Using the documented hex PSM makes construction revert because `Minter2` calls the unsupported `gemJoin()` function on the GemJoin contract. However, the constructor otherwise checks only that dependencies are nonzero and does not validate their relationships:

```
if (
 admin_ == address(0) || address(usdt_) == address(0)
 || address(unit_) == address(0) || address(usdd_) == address(0)
 || address(psm_) == address(0) || address(jUsdd_) == address(0)
 || address(stakedUnit_) == address(0)
) revert ZeroAddress();

USDD = usdd_;
```

```
PSM = psm_;
jUSDD = jUsdd_;

USDT.forceApprove(psm_.gemJoin(), type(uint256).max);
USDD.forceApprove(address(jUsdd_), type(uint256).max);
```

A zero-output mint becomes possible if the deployer corrects the PSM address but retains the README's valid legacy USDD Base58 address. This mixed configuration can pass construction because legacy USDD is a functioning token.

During minting, the active PSM takes the user's USDT and transfers active USDD to Minter2. Minter2 instead measures the configured legacy token's balance:

```
USDT.safeTransferFrom(msg.sender, address(this), assets);

uint256 usddBefore = USDD.balanceOf(address(this));
PSM.sellGem(address(this), assets);
uint256 usddReceived = USDD.balanceOf(address(this)) - usddBefore;

if (usddReceived > 0 && jUSDD.mint(usddReceived) ≠ 0) {
 revert OperationFailed();
}
return usddReceived / 1e12;
```

The measured delta is zero, so `_mintInternal` returns zero. The public `mint` function accepts this result and emits `Minted(account, 0)` without issuing UNIT. The active USDD remains unaccounted for in Minter2 and requires administrator intervention through `withdraw` to recover or refund.

The test named `ForkMainnet.t.sol` does not detect these incompatibilities because it replaces the listed addresses with mock bytecode instead of interacting with deployed TRON contracts.

Recommendations:

It is recommended to maintain one canonical deployment registry containing verified Base58 and hex representations:

- USDT: `TR7NHqjeKQxGTCi8q8ZY4pL8otSzgjLj6t` / `0xa614f803B6FD780986A42c78Ec9c7f77e6DeD13C`
- Active USDD: `TXDk8mbtRbXeYuMNS83CfKPaYYT8XWv9Hz` / `0xE91A7411e56Ce79E83570570f49B9FC35B7727c5`
- Active PSM: `TBXW4hS5KYjjbJXDpnrPf4zhkLwrpUjbyz` / `0x1113ae08a16489a7b76f2ccc52290ab54e2783d8`

- USDT GemJoin: `TSUYvQ5tdd3DijCD1uGunGLpftHuSZ12sQ` / `0xB50Eb419ebeBA06c80Df5e9AaeC494Cef4297879`
- jUSDD: `TKFRELGGoRgiayhwJTNNLqCNjFoLBh3Mnf` / `0x65c9feDE72Ba73CD1B0DCA2A974C070153dC6FCB`

Validate integration relationships during construction:

```
address gemJoin = psm_.gemJoin();

if (
 stakedUnit_.asset() ≠ address(unit_)
 || IUnderlying(address(jUsdd_)).underlying() ≠ address(usdd_)
 || IPSMMetadata(address(psm_)).usdd() ≠ address(usdd_)
 || IGemJoin(gemJoin).gem() ≠ address(usdt_)
) revert InvalidIntegration();
```

Also validate expected token decimals, revert whenever positive input produces zero UNIT, and add an integration smoke test against deployed TRON bytecode rather than etched mocks.

UNIT: Resolved with `@875a656f4c ...` and `@cfa5ca199f ...` - by preventing zero-output mints, hardcoding verified TRON dependencies, and validating token decimals and integration relationships.

Zenith: Verified.

[I-2] `MINTER_ROLE` unnecessarily includes arbitrary third-party burning

SEVERITY: Informational	IMPACT: Informational
STATUS: Resolved	LIKELIHOOD: Low

Target:

- src/Unit.sol#L23-L29

Description:

`Unit` uses the same role for unlimited issuance and burning tokens from arbitrary accounts:

```
function mint(
 address to,
 uint256 assets
) external onlyRole(MINTER_ROLE) {
 _mint(to, assets);
}

function burn(
 address from,
 uint256 assets
) external onlyRole(MINTER_ROLE) {
 _burn(from, assets);
}
```

A minter can destroy UNIT from any address without allowance or authorization from the holder. This authority is broader than required for issuance and gives every present or future minter forced-burning power.

Forced burning differs from `confiscate`: it destroys the holder's value instead of transferring it. The operation emits the standard ERC-20 `Transfer(from, address(0), amount)` event but no policy-specific event distinguishing an administrative forced burn from an ordinary redemption burn.

A minter can also burn UNIT held by `StakedUnit`, reducing vault backing without cancelling corresponding sUNIT shares. Granting `MINTER_ROLE` to `StakedUnit` itself does not enable this action because the current vault has no function capable of calling `Unit.burn`.

Nevertheless, `README.md` and the main integration test assign `MINTER_ROLE` to `StakedUnit`, even though it never calls either privileged Unit function. The integration test additionally grants the role directly to the administrator. The repository contains no production deployment script, so actual production role membership is unverified.

The documented administrator and intended minters are trusted, and a compromised minter can already cause severe damage through unlimited issuance. This is therefore a least-privilege and operational-containment concern rather than an unprivileged exploit.

Recommendations:

It is recommended to remove the documented and tested `StakedUnit` role grant and assign issuance authority only to contracts that require it.

Separate issuance from ordinary burning and prefer self-burning or allowance-authorized burning:

```
bytes32 public constant ISSUER_ROLE =
 keccak256("ISSUER_ROLE");

function mint(
 address to,
 uint256 amount
) external onlyRole(ISSUER_ROLE) {
 _mint(to, amount);
}

function burn(uint256 amount) external {
 _burn(msg.sender, amount);
}

function burnFrom(
 address from,
 uint256 amount
) external {
 _spendAllowance(from, msg.sender, amount);
 _burn(from, amount);
}
```

Under this model, direct UNIT redemption requires the user to approve `Minter2`, which then transfers the UNIT to itself and burns its own balance. Adopt this change only if the additional approval requirement is acceptable.

Use the existing `confiscate` function for regulatory seizure. Add a separate forced-burn role and dedicated event only if permanent administrative destruction is an explicit policy requirement.

Update deployment documentation and integration setup to list only necessary role holders, and monitor `RoleGranted` and `RoleRevoked` events for unexpected or stale assignments.

UNIT: Resolved with `@ef415ac333 ...` - by limiting MINTER_ROLE to issuance and replacing arbitrary burns with self- and allowance-authorized burning.

Zenith: Verified.

[I-3] Signer reauthorization can reactivate outstanding permits

SEVERITY: Informational	IMPACT: Informational
STATUS: Acknowledged	LIKELIHOOD: Low

Target:

- src/Minter2.sol#L35-L49
- src/Minter2.sol#L96-L123
- src/Minter2.sol#L188-L190

Description:

Mint and redeem permits share one sequential nonce per account, while signer authorization is checked only at execution:

```
keccak256(
 abi.encode(
 MINT_TYPEHASH,
 msg.sender,
 assets,
 stake,
 _useNonce(msg.sender),
 deadline
 )
);

keccak256(
 abi.encode(
 REDEEM_TYPEHASH,
 msg.sender,
 assets,
 unstake,
 _useNonce(msg.sender),
 deadline
 )
);

function _checkPermit(
 bytes32 digest,
```

```
  uint256 deadline,
  bytes calldata signature
 ) private view {
  if (block.timestamp > deadline) {
  revert PermitExpired();
  }

  _checkRole(
  SIGNER_ROLE,
  digest.recover(signature)
  );
 }
```

Revoking `SIGNER_ROLE` invalidates signatures produced by that signer while the role remains revoked. If the same address is subsequently reauthorized, any signature that is still unexpired and uses the account's current nonce becomes valid again because the signed message contains no signer-authorization epoch.

The shared nonce also serializes mint and redeem permits. If the signer pre-signs multiple sequential permits, a higher-nonce permit cannot execute until the preceding nonce is consumed. The account has no function for skipping that nonce. Separate permits signed with the same nonce are intentionally mutually exclusive.

These nonce semantics are standard replay protection and are documented by the protocol. Because the signed account is bound to `msg.sender`, another account cannot execute or cancel the permit. User-controlled nonce cancellation would therefore primarily provide queue management rather than protection against third-party execution.

The contract also accepts any signer-selected deadline without enforcing a maximum validity window. This is an authorization-policy choice because the trusted signer controls the signed deadline.

The primary concern is therefore the possible reactivation of outstanding signatures after signer reauthorization. The remaining limitations are operational and usability considerations.

Recommendations:

It is recommended to use short permit validity periods and never reauthorize an address whose signing key may have been compromised.

If signer addresses must be reauthorized, include an authorization epoch in both typed-data structures and increment it automatically whenever the `SIGNER_ROLE` membership changes. A global epoch is simpler but invalidates outstanding signatures from every signer; a per-signer epoch provides more granular invalidation.

Add separate keyed nonce channels only if mint and redeem permits must progress independently. If multiple permits of the same type must remain concurrently executable,

use unordered nonces, such as unique salts or nonce bitmaps, rather than another sequential nonce.

An optional account-controlled cancellation function may advance the relevant nonce to skip queued permits. To enforce a maximum validity window, include a signed issuance or validity-start timestamp and reject permits whose signed validity interval exceeds the configured limit.

UNIT: Acknowledged.

- The off-chain signer service generates signatures with short validity windows (short deadlines).
- Operational security guidelines dictate that compromised signer keys are permanently revoked and never reauthorized with SIGNER_ROLE. Fresh key pairs are provisioned instead.

[I-4] `StakedUnit` blocks share transfers but retains delegated ERC-4626 exits

SEVERITY: Informational	IMPACT: Informational
STATUS: Acknowledged	LIKELIHOOD: Low

Target:

- src/StakedUnit.sol#L8-L26
- src/Minter2.sol#L96-L130

Description:

`StakedUnit` prevents direct share transfers by rejecting every update between nonzero addresses:

```
function _update(
 address from,
 address to,
 uint256 value
) internal override {
 if (
 from ≠ address(0)
 && to ≠ address(0)
 ) {
 revert NonTransferable();
 }

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

The contract nevertheless retains the standard inherited ERC-4626 entry, approval, and exit behavior:

- `deposit` and `mint` may issue new shares to an arbitrary receiver;
- an owner may burn shares and send the released UNIT to any receiver;
- an approved operator may burn an owner's shares and redirect the UNIT, up to the granted allowance;
- a user receiving sUNIT through `Minter2.mint( ... , stake=true)` may subsequently withdraw directly without another signer authorization.

These operations do not transfer existing sUNIT. Depositing for another receiver is a voluntary assignment when entering the vault, while withdrawal burns the shares before transferring the underlying asset. Delegated withdrawal also requires an explicit sUNIT allowance and is not an unauthorized extraction.

Consequently, non-transferability applies only to the share-token layer. It does not make the economic claim permanently owner-bound or make the signed `stake` selection persistent. Permanent owner binding would additionally be impossible while holders can withdraw freely transferable UNIT.

This behavior is consistent with the documented design, which states that transfers between nonzero addresses revert while minting and burning operate normally. The documentation does not claim that staking positions are economically soulbound or that every future exit requires signer authorization. This is therefore an informational policy and integration consideration unless stronger restrictions were intended.

The implementation also rejects zero-value peer transfers. Contracts that support recovery only through ERC-20 transfers cannot recover sUNIT minted to them. For sUNIT held by `Minter2`, recovery instead requires the administrator to invoke `StakedUnit.redeem` through `executeCall`; other contracts without arbitrary-call or ERC-4626 support may permanently lock the shares.

Recommendations:

It is recommended to explicitly document that:

- only direct sUNIT transfers are prohibited;
- sUNIT allowances authorize delegated ERC-4626 withdrawals;
- an approved operator may select the UNIT receiver;
- shares may be minted directly to another receiver;
- holders may withdraw to UNIT without signer authorization;
- contract recipients must support ERC-4626 redemption to recover their shares.

Allow zero-value transfers for improved ERC-20 compatibility without enabling economic transfers:

```
function _update(
 address from,
 address to,
 uint256 value
) internal override {
 if (
 value ≠ 0
 && from ≠ address(0)
 && to ≠ address(0)
 ) {
 revert NonTransferable();
```

```
  }

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

If delegated redirection is not intended, constrain `caller`, `owner`, and `receiver` in the vault exit path. Preserve an explicit `Minter2` exception or redesign its unstaking flow, because signed sUNIT redemption currently relies on an allowance granted to `Minter2`.

If all staking and unstaking must be signer-controlled, restrict both vault entry and exit to an authorized gateway such as `Minter2`. This materially reduces ERC-4626 composability and should be implemented only as an explicit product-policy change.

UNIT: Partially resolved with `@795ec6e9c0 ...` by allowing zero-value transfers, and further acknowledged.

- The restriction on sunitUSD is intentionally scoped only to direct peer-to-peer share transfers to keep secondary trading focused on liquid unitUSD.
- Retaining standard ERC-4626 entry, exit, allowance-based delegated withdrawals, and receiver specifications is fully intentional by protocol design to preserve DeFi composability and facilitate Minter2.redeem(..., unstake=true) operations.

Zenith: Verified and acknowledged.

[I-5] Small mints can consume USDT while issuing zero UNIT

SEVERITY: Informational	IMPACT: Informational
STATUS: Resolved	LIKELIHOOD: Low

Target:

- Minter2.sol#L164-L172

Description:

`_mintInternal()` rounds `usddReceived / 1e12` down. With a positive PSM entry fee and a sufficiently small USDT input, Minter2 can successfully consume the USDT and wrap the resulting USDD while `unitToMint == 0`.

The affected value is bounded to sub-micro-UNIT dust under ordinary fee settings. The operation nevertheless succeeds rather than rejecting a zero-output conversion.

Recommendations:

Require `unitToMint > 0`

UNIT: Resolved with `@9aedc0252a ...`

- Added check in Minter2.mint() requiring assets != 0 && unitToMint != 0, reverting with InsufficientOutput if input produces zero unitUSD.
- Added guard in Minter2._mintInternal() reverting with OperationFailed if assets > 0 && unitToMint == 0.

Zenith: Verified.

[I-6] `executeCall()` can report success when status-code-based external operations failed

SEVERITY: Informational	IMPACT: Informational
STATUS: Resolved	LIKELIHOOD: Low

Target:

- Minter2.sol#L157-L162

Description:

`executeCall()` checks only whether the low-level call reverted and discards returned data.

Compound/JustLend-style functions may signal operational failure by returning a nonzero error code while the EVM/TVM call itself succeeds. In that case `executeCall()` completes successfully despite the intended administrative operation failing.

Risk Analysis

Only the admin can invoke the function and state can be verified afterward, limiting impact. The risk is primarily incorrect emergency/operational tooling during incidents.

Recommendations:

Return the external call's `bytes returndata` to the caller and have administrative tooling validate any protocol-specific status code.

UNIT: Resolved with `@16f404c3a0 ...`

- Updated Minter2.executeCall() to return bytes memory returndata allowing callers to inspect returned error codes or data.
- Marked executeCall() as payable to support value-bearing administrative calls when necessary.

Zenith: Verified. No `receive()/fallback()` on Minter2 in this fix but it was independently fixed in another issue, so refunded TRX won't revert in the final commit.