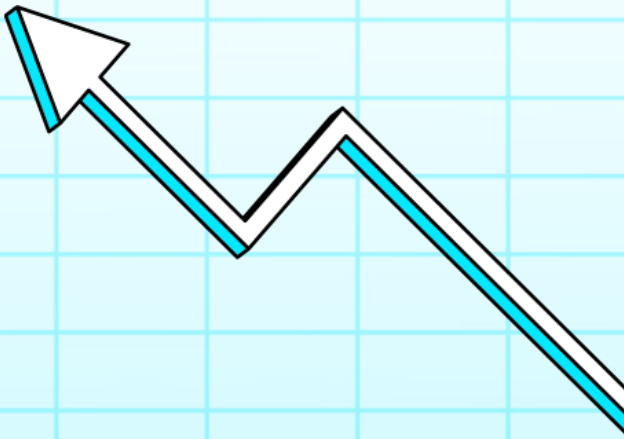
shieldify



Prime Vaults

SECURITY REVIEW

Date: 15 December 2025

CONTENTS

1. About Shieldify

Positioned as the first hybrid Web3 Security company, Shieldify shakes things up with a unique subscription-based auditing model that entitles the customer to unlimited audits within its duration, as well as top-notch service quality thanks to a disruptive 6-layered security approach. The company works with very well-established researchers in the space and have secured multiple millions in TVL across protocols, also can audit codebases written in Solidity, Vyper, Rust, Cairo, Move and Go.

Learn more about us at shieldify.org.

2. Disclaimer

This security review does not guarantee bulletproof protection against a hack or exploit. Smart contracts are a novel technological feat with many known and unknown risks. The protocol, which this report is intended for, indemnifies Shieldify Security against any responsibility for any misbehavior, bugs, or exploits affecting the audited code during any part of the project's life cycle. It is also pivotal to acknowledge that modifications made to the audited code, including fixes for the issues described in this report, may introduce new problems and necessitate additional auditing.

3. About Prime Vaults

Prime Vaults redefines on-chain saving by combining Principal protection with a Guaranteed Minimum Yield, ensuring every supported asset earns a baseline return that is always equal to or higher than its benchmark rate on Aave. It functions as a smart saving account designed to deliver the simplicity and reliability of traditional savings while leveraging the composability of decentralized finance.

When users deposit assets, their balance is conceptually divided into two components: Principal and Yield.

- The Principal represents the user's protected capital. It is deployed across curated yield strategies but is safeguarded by the IL Reserve Fund, which absorbs volatility arising from impermanent loss or strategy fluctuations. This ensures that the user principal remains intact under all supported market conditions.
- The Yield credited to users is governed primarily by the protocol's Guaranteed Minimum Yield framework. Instead of relying solely on fluctuating strategy performance, Prime Vaults sets a min APY for each asset type and guarantees that depositors receive no less than this rate. Users may claim their accrued yield at any time.

To deliver both principal protection and a guaranteed minimum yield, Prime Vaults is built upon a unified liquidity and yield architecture. This system departs from the traditional isolated-asset model and enables efficient capital pairing, stable return generation, and consistent yield distribution across all depositors.

Learn more about Prime Vaults' concept and the technicalities behind it: here

4. Risk Classification

Severity	Impact: High	Impact: Medium	Impact: Low
Likelihood: High	Critical	High	Medium
Likelihood: Medium	High	Medium	Low
Likelihood: Low	Medium	Low	Low

4.1 Impact

- **High** – results in a significant risk for the protocol's overall well-being. Affects all or most users
- **Medium** – results in a non-critical risk for the protocol affects all or only a subset of users, but is still unacceptable
- **Low** – losses will be limited but bearable – and covers vectors similar to griefing attacks that can be easily repaired

4.2 Likelihood

- **High** – almost certain to happen and highly lucrative for execution by malicious actors
- **Medium** – still relatively likely, although only conditionally possible
- **Low** – requires a unique set of circumstances and poses non-lucrative cost-of-execution to rewards ratio for the actor

5. Security Review Summary

The security review lasted 5 days with a total of 40 hours dedicated to the audit by the Shieldify team.

Overall, the code is well-written. The audit report contributed by identifying one High, four Medium and three Low severity findings. They're mainly related to strategy lifecycle and withdrawal safety, with additional issues around denial-of-service risks, incorrect token handling, insufficient validation of swap parameters, and incomplete pause controls.

The Beraji team has done a great job with their test suite and provided support and responses to all of the questions that the Shieldify researchers had.

5.1 Protocol Summary

Project Name	Prime Vaults
Repository	prime-vaults-strategy
Type of Project	DeFi, Vault, Yield
Security Review Timeline	5 days
Review Commit Hash	b722f446d14f0ff4fd600dab80a16ac8b11b413d
Fixes Review Commit Hash	ec4fef8ea07ba68f64352ca357a81793074f52c7

5.2 Scope

The following smart contracts were in the scope of the security review:

File	nSLOC
contracts/StrategyRegistry.sol	87
contracts/PrimeStrategy.sol	113
contracts/StrategyManager.sol	288
Total	**488**

6. Findings Summary

The following number of issues have been identified, sorted by their severity:

- **High** issues: **1**
- **Medium** issues: **4**
- **Low** issues: **3**
- **Info** issues: **3**

ID	Title	Severity	Status
[H-01]	Removed Strategy Can Bypass Removal and Block Withdrawals	High	Fixed
[M-01]	Denial of Service via Storage Bloating from Unrestricted Deposits	Medium	Fixed
[M-02]	Failure in Single Strategy Blocks All User Withdrawals	Medium	Fixed
[M-03]	Pair Allocation Caps Can Be Assigned to the Wrong Token	Medium	Fixed
[M-04]	Swap Calldata Is Not Validated Against `SwapParams`, Allowing `tokenOut` / `minAmountOut` Mismatches	Medium	Fixed
[L-01]	`StrategyManager` Inherits Pausable But Lacks Pause/Unpause Functionality	Low	Fixed
[L-02]	Harvest Function Denial of Service Due to Single Strategy Revert	Low	Fixed
[L-03]	Withdraw Priority Updates Lack Validation of Allocation Keys	Low	Fixed
[I-01]	Missing Zero Amount Validation in Deposit and Withdraw Functions	Info	Fixed
[I-02]	Missing Event Emissions for Administrative Parameter Updates	Info	Fixed
[I-03]	Missing Zero Address Checks	Info	Fixed

7. Findings

[H-01] Removed Strategy Can Bypass Removal and Block Withdrawals

Severity

High Risk

Description

The `StrategyRegistry.removeStrategy()` function allows the owner to permanently remove a strategy (the context). However, removal is implemented via `delete strategies[stra]`, which resets `kind` to its default enum value (`StrategyKind.SingleAsset == 0`) and sets `active` to `false`, while `PrimeStrategy` ignores `active` and only validates `kind` inside `_withdrawSingle()` / `_withdrawPair()` (the problem). This leads to inconsistent behavior during `PrimeStrategy.withdraw()` when iterating `withdrawPriority`: for `SingleAsset` strategies, a removed strategy appears as `SingleAsset` (kind == 0), so `_withdrawSingle()` passes the kind check and can still call `withdraw()` on a strategy address that was removed from the registry, allowing a bypass; for PairAsset strategies, a removed strategy also appears as `SingleAsset` (kind == 0), so `_withdrawPair()` fails the kind check and reverts, which bubbles up and reverts the entire user `withdraw()`, causing a DoS.

Location of Affected Code

File: contracts/StrategyRegistry.sol#L80-L95

```
function removeStrategy(address stra) external onlyOwner {
    // code
    // @audit Deleting resets (kind, active) to defaults (0, false)
    delete strategies[stra];
    // code
}
```

File: contracts/PrimeStrategy.sol

```
function withdraw(address token, uint256 amount) external nonReentrant {
    // code
    if (shortfall > 0) {
        for (uint256 i; i < withdrawPriority.length && shortfall > 0;) {
            WithdrawPriority memory wq = withdrawPriority[i];
            uint256 withdrawn = 0;

            // @audit If a removed Pair strategy is reached,
               _withdrawPair reverts
            if (wq.kind == StrategyKind.SingleAsset) {
                withdrawn = _withdrawSingle(wq.allocKey, shortfall, token
                   );
            } else {
                withdrawn = _withdrawPair(wq.allocKey, shortfall, token);
            }
```

```
            shortfall = withdrawn >= shortfall ? 0 : shortfall -
                withdrawn;
            unchecked {
                ++i;
            }
        }
        // code
    }
    // code
}

function _withdrawSingle(bytes32 key, uint256 want, address _token)
    internal returns (uint256 withdrawn) {
    // code
    (StrategyKind kind,) = IStrategyRegistry(strategyRegistry).strategies
        (alloc.strategy);

    // @audit Removed strategy returns kind=0 (SingleAsset), passing this
        check
    if (kind != StrategyKind.SingleAsset) revert
        VAULT__INVALID_STRATEGY_KIND();

    // ... calls ISingleAssetStrategy(alloc.strategy).withdraw(...)
}

function _withdrawPair(bytes32 key, uint256 want, address _token)
    internal returns (uint256 withdrawn) {
    // code
    (StrategyKind kind,) = IStrategyRegistry(strategyRegistry).strategies
        (alloc.strategy);

    // @audit Removed strategy returns kind=0 (SingleAsset), reverting
        here
    if (kind != StrategyKind.PairAsset) revert
        VAULT__INVALID_STRATEGY_KIND();
    // code
}
```

Impact

The inconsistency between strategy removal and the withdrawal logic leads to two distinct impacts:

- `SingleAsset` strategies that have been explicitly removed by the protocol owner can still be interacted with during withdrawals. If a strategy was removed due to malicious behavior or a bug, continuing to execute withdrawals against it puts user funds at risk and ignores the administrator's intent to disconnect the strategy.
- If a `PairAsset` strategy is removed but remains in the withdrawal queue, it triggers a revert in the withdrawal flow due to the mismatched `kind`. This causes the entire `withdraw()` transaction to fail for users, effectively locking funds until the configuration is manually corrected.

Proof of Concept

1. Owner removes a strategy via `StrategyRegistry.removeStrategy()`, which executes `delete strategies[stra]`.
2. The deleted entry now returns `kind = StrategyKind.SingleAsset(0)` and `active = false`.
3. A user calls `PrimeStrategy.withdraw()` and the vault calculates a `shortfall`.
4. The withdrawal loop iterates `withdrawPriority`.
5. **Bypass path**: if the entry is treated as `SingleAsset`, `_withdrawSingle()` sees `kind == 0` and proceeds to call `withdraw()` on the removed strategy address.
6. **DoS path**: if the entry is treated as `PairAsset`, `_withdrawPair()` sees `kind == 0`, reverts `VAULT__INVALID_STRATEGY_KIND()`, and the entire `withdraw()` reverts.

Recommendation

In `_withdrawSingle()` and `_withdrawPair()`, fetch and enforce the `active` flag from the registry and skip inactive/removed strategies by returning `0` (so the withdrawal loop can continue) or use the try catch block to handle the revert.

- Read `(StrategyKind kind, bool active)` = `IStrategyRegistry(strategyRegistry).strategies(alloc.strategy);`
- If `!active`, return `0`
- Keep the `kind` checks as-is for active strategies

Optionally, wrap external strategy withdrawals in `try/catch` to ensure a single bad/paused strategy cannot revert the whole `withdraw()` flow.

Team Response

Fixed.

[M-01] Denial of Service via Storage Bloating from Unrestricted Deposits

Severity

Medium Risk

Description

The `deposit()` function lacks a minimum deposit threshold and uses an O(n) linear search via `_checkExists()` to verify token existence in the `depositedTokens` array. An attacker can deploy numerous tokens and deposit minimal amounts (e.g., 1 wei) for each, causing unbounded array growth. As the array grows, `_checkExists()` becomes increasingly expensive, eventually causing denial of service when gas costs exceed block limits.

Location of Affected Code

File: contracts/PrimeStrategy.sol

```
function deposit(address token, uint256 amount) external nonReentrant {
    // @audit missing minimum amount check allows array bloat with very
       low amounts and denial of service...
    IERC20(token).safeTransferFrom(msg.sender, address(this), amount);
    deposited[token] += amount;
    depositor[msg.sender][token] += amount;
    if (!_checkExists(token)) {
        depositedTokens.push(token);
    }
    emit Deposited(token, amount, msg.sender);
}

function _checkExists(address token) internal view returns (bool exists)
   {
    for (uint256 i = 0; i < depositedTokens.length; i++) {
        if (depositedTokens[i] == token) {
            exists = true;
            break;
        }
    }
}
```

Impact

An attacker can bloat the contract state by depositing minimal amounts (e.g., 1 wei) or even zero amounts for thousands of different tokens. The `depositedTokens` array grows unboundedly, causing denial of service on deposits as `_checkExists()` consumes increasing amounts of gas with each new entry. Eventually, legitimate deposits will exceed the block gas limit or become prohibitively expensive. Additionally, each new token entry permanently increases the contract's storage size, consuming blockchain resources and making the contract state increasingly expensive to read and write. Functions that iterate over `depositedTokens` become unusable as the array grows, further degrading the contract's functionality.

Proof of Concept

1. Attacker deploys a token factory contract that can create many ERC20 tokens cheaply.
2. Attacker calls `PrimeStrategy.deposit(tokenAddress, 1)` repeatedly with 10,000 different token addresses, spending minimal gas and token amounts (only 1 wei per token or zero amount transfers).
3. The `depositedTokens` array grows to 10,000+ entries, each consuming storage slots and bloating the contract state.
4. A legitimate user attempts to deposit USDC: `deposit(USDC, 1000)`.
5. The transaction reverts with "Out of Gas" because `_checkExists(USDC)` must iterate through all 10,000+ entries in the bloated array, consuming excessive gas.
6. The attacker has successfully bloated the contract state with minimal cost, making it unusable for legitimate users.

Recommendation

Implement a minimum deposit threshold and replace the O(n) array-based existence check with an O(1) mapping-based approach:

```
mapping(address => bool) public tokenTracked;

function deposit(address token, uint256 amount) external nonReentrant {
    require(amount >= MIN_DEPOSIT_AMOUNT, "Amount too small");

    IERC20(token).safeTransferFrom(msg.sender, address(this), amount);
    deposited[token] += amount;
    depositor[msg.sender][token] += amount;

    if (!tokenTracked[token]) {
        tokenTracked[token] = true;
        depositedTokens.push(token);
    }

    emit Deposited(token, amount, msg.sender);
}
```

Team Response

Fixed.

[M-02] Failure in Single Strategy Blocks All User Withdrawals

Severity

Medium Risk

Description

Failure in a single strategy blocks all user withdrawals, causing a temporary DoS.

The `PrimeStrategy.withdraw()` function iterates through the `withdrawPriority` queue to fetch funds if the idle balance is insufficient. It attempts to withdraw from each strategy in order.

If any strategy in the `withdrawPriority` list reverts (e.g., due to being paused, having insufficient liquidity, or a bug in the strategy code), the entire user withdrawal transaction will revert. This effectively freezes user funds until the manager manually updates the `withdrawPriority` to remove the failing strategy.

Location of Affected Code

File: contracts/PrimeStrategy.sol#L67-L90

```
function withdraw(address token, uint256 amount) external nonReentrant {
    // code
    if (shortfall > 0) {
        for (uint256 i; i < withdrawPriority.length && shortfall > 0; ) {
            // code
            // @audit If _withdrawSingle reverts, the whole tx reverts
            if (wq.kind == StrategyKind.SingleAsset) {
                withdrawn = _withdrawSingle(wq.allocKey, shortfall, token
                    );
            }
            // code
        }
        // code
    }
}
```

Impact

Users are unable to access their funds on demand if a single integrated strategy fails. While the Manager can fix this by updating the priority queue, it creates a period of illiquidity and dependence on manual intervention.

Proof of Concept

The issue is evident in the `withdraw()` logic, where `_withdrawSingle()` is called inside a loop. Since there is no error handling (try/catch) around the external call to the strategy (via `_withdrawSingle()`), any revert bubbles up and causes the entire withdraw transaction to fail.

1. User calls `withdraw()`.
2. Contract calculates `shortfall`.
3. Loop starts iterating `withdrawPriority`.
4. `_withdrawSingle()` is called for the first strategy.
5. Strategy reverts (e.g. paused).
6. The withdrawal transaction reverts.

Recommendation

Wrap the withdrawal calls to strategies in a `try/catch` block. If a strategy reverts, skip it and proceed to the next one in the priority queue. Alternatively, ensure `_withdrawSingle()` / `_withdrawPair()` checks for potential failure conditions before calling the external contract, though `try/catch` is more robust.

Team Response

Fixed.

[M-03] Pair Allocation Caps Can Be Assigned to the Wrong Token

Severity

Medium Risk

Description

The `addPairAllocation()` function normalizes `(tokenA, tokenB)` into ascending address order (and `_pairKey()` relies on the same ordering), but it does **not** normalize the corresponding cap inputs. If the manager provides tokens in descending order (`tokenA > tokenB`), `capA` / `capB` can be stored against the wrong token, swapping the intended per-token caps for that pair.

Location of Affected Code

File: contracts/StrategyManager.sol

```
function addPairAllocation(address tokenA, address tokenB, uint256 capA,
    uint256 capB, address stra) external onlyManager {
     // code
     bytes32 key = _pairKey(stra, tokenA, tokenB);
     PairAllocation storage alloc = pairAllocs[key];

     if (alloc.exists) {
          // @audit Caps are also not reordered on update
          alloc.capWantedA = capA;
          alloc.capWantedB = capB;
     } else {
          // @audit Tokens are reordered but caps are NOT
          (address a, address b) = tokenA < tokenB
               ? (tokenA, tokenB)
               : (tokenB, tokenA);
          pairAllocs[key] = PairAllocation(stra, a, b, 0, 0, capA, capB,
             true);
          // code
     }
}
```

Impact

If a privileged manager configures a pair allocation with tokens passed in descending address order (`tokenA > tokenB`), the stored caps can become associated with the wrong token. As a result, downstream flows like `executeAllocPair()`, `_handleDeposit()`, and `_handleWithdrawal()` will operate with swapped intended amounts for that pair, potentially leading to persistent over/under-funding of a strategy and degraded performance/risk management for any affected pair-based allocation.

Proof of Concept

Consider the Ethereum mainnet: - USDC: `0xA0b86991c6218b36c1d19D4a2e9Eb0cE3606eB48` - USDT: `0xdAC17F958D2ee523a2206206994597C13D831ec7`

Manager intends: 1000 USDT, 2000 USDC

```
// Manager calls with USDT first (larger address)
addPairAllocation(USDT, USDC, 1000, 2000, strategy);

// Internal processing:
// 1. key = _pairKey(strategy, USDT, USDC)
//    Since USDT > USDC: keccak256(strategy, USDC, USDT)
//
// 2. Token reordering: (a, b) = (USDC, USDT) since USDC < USDT
//
// 3. Storage creation:
//    PairAllocation(strategy, USDC, USDT, 0, 0, 1000, 2000, true)
//                             ^a     ^b          ^capA ^capB
//
// Result:
//   tokenA = USDC, capWantedA = 1000 (but meant for USDT!)
//   tokenB = USDT, capWantedB = 2000 (but meant for USDC!)
```

When `executeAllocPair()` runs, it deposits: - 1000 USDC (should have been 2000) - 2000 USDT (should have been 1000)

This is the **exact opposite** of the intended allocation.

Recommendation

Reorder the cap values along with the tokens:

```
function addPairAllocation(address tokenA, address tokenB, uint256 capA,
   uint256 capB, address stra) external onlyManager {
    (StrategyKind kind, bool active) = IStrategyRegistry(strategyRegistry
       )
        .strategies(stra);
    if (!active) revert VAULT__STRATEGY_NOT_ACTIVE();
    if (kind != StrategyKind.PairAsset) revert
       VAULT__INVALID_STRATEGY_KIND();

    bytes32 key = _pairKey(stra, tokenA, tokenB);
    PairAllocation storage alloc = pairAllocs[key];

    if (alloc.exists) {
        // Reorder caps consistently on update
        if (tokenA < tokenB) {
            alloc.capWantedA = capA;
            alloc.capWantedB = capB;
        }
```

```
    else {
            alloc.capWantedA = capB;
            alloc.capWantedB = capA;
        }
    } else {
        // Reorder both tokens AND caps together
        (address a, address b, uint256 cA, uint256 cB) = tokenA < tokenB
            ? (tokenA, tokenB, capA, capB)
            : (tokenB, tokenA, capB, capA);

        pairAllocs[key] = PairAllocation(stra, a, b, 0, 0, cA, cB, true);
        pairKeys.push(key);
        withdrawPriority.push(WithdrawPriority(StrategyKind.PairAsset,
           key));
    }
}
```

Team Response

Fixed.

[M-04] Swap Calldata Is Not Validated Against `SwapParams`, Allowing `tokenOut` / `minAmountOut` Mismatches

Severity

Medium Risk

Description

The `swap()` function allows the vault manager (via `onlyManager()`) to execute a swap by forwarding a raw calldata blob (`params.data`) to an external target (`params.to`). The calldata is expected to be KyberSwap-compatible (it is gated by `KX_SWAP_SELECTOR`). At the same time, it also accepts an explicit `SwapParams` struct (e.g., `tokenIn`, `tokenOut`, `amountIn`, `minAmountOut`) that appears to describe and constrain the intended swap.

However, the function does not validate that the raw `params.data` it forwards is consistent with the explicit `SwapParams`. It decodes `params.data` to read `KxOutput`, but only checks the `receiver` field and discards the decoded `KxInput` entirely.

As a result: - `params.tokenOut` is not checked against the decoded `output.token`. - `params.minAmountOut` is not checked against the decoded `output.minAmountOut`. - `params.tokenIn` / `params.amountIn` are not checked against the decoded `KxInput` (since it is decoded but ignored).

Additionally, `swap()` does not validate that the vault's `params.tokenOut` balance actually increases (i.e., that the swap produced `tokenOut` at all).

Location of Affected Code

File: contracts/StrategyManager.sol#L430-L452

```
function swap(SwapParams calldata params) external onlyManager
    nonReentrant {
     // code
     require(bytes4(params.data) == KX_SWAP_SELECTOR, 'KX: bad selector');
     bytes memory payload = params.data[4:];

     // @audit KxInput is decoded but ignored; output fields are not
        validated vs `params`.
     (, KxOutput memory output, , ) = abi.decode(payload, (KxInput,
        KxOutput, KxSwapData, KxFee));
     require(output.receiver == address(this), 'bad receiver');

     // code
     (bool ok, bytes memory res) = params.to.call{value: params.value}(
        params.data);
     // code
}
```

Impact

- Integrations may treat `params.minAmountOut` as a meaningful constraint, but the contract does not ensure the forwarded KyberSwap calldata enforces an equal-or-stronger minimum output.
- The call can succeed while the vault receives a different token than `params.tokenOut`, meaning the vault's `tokenOut` balance increase can be `0` even though the swap was "declared" as `tokenOut` in `SwapParams` (and recorded in `swapped[tokenIn]`).

Proof of Concept

1. **Scenario**: The manager intends to swap USDC for WETH with `params.minAmountOut = 1 ether` and `params.tokenOut = WETH`.
2. **Input**: The manager constructs `SwapParams` with the expected values.
3. **Bug/misconfiguration**: The off-chain component that builds `params.data` provides calldata where:
    - `output.token` is not WETH (e.g., DAI), and/or
    - `output.minAmountOut` is set to a much smaller number (e.g., `0`).
4. **Execution**:
    - `swap()` only checks the selector and `output.receiver`.
    - It does not verify `output.token == params.tokenOut` or that `output.minAmountOut >= params.minAmountOut`.
    - It records the swap pair using `params.tokenOut`, even if the forwarded calldata swaps into a different token (so `params.tokenOut` may not increase at all).

5. **Result**: The transaction can succeed while violating the explicit intent expressed in `SwapParams`, and internal/off-chain tracking can become inconsistent with what actually happened.

Recommendation

Treat `SwapParams` as the source of truth and validate that the forwarded calldata is consistent with it. At minimum: - Decode and compare `KxInput` / `KxOutput` against `params.tokenIn`, `params.tokenOut`, and `params.amountIn`. - Enforce that the minimum amount out encoded in `params.data` is at least `params.minAmountOut`. - Optionally, validate the actual `tokenOut` balance delta is at least `params.minAmountOut` (after enforcing `output.token == params.tokenOut`), so a successful call implies `tokenOut` increased.

```
function swap(SwapParams calldata params) external onlyManager
    nonReentrant {
     // code
     require(bytes4(params.data) == KX_SWAP_SELECTOR, 'KX: bad selector');
     bytes memory payload = params.data[4:];

     (KxInput memory input, KxOutput memory output, , ) =
         abi.decode(payload, (KxInput, KxOutput, KxSwapData, KxFee));

     require(output.receiver == address(this), 'bad receiver');
     require(input.token == params.tokenIn, 'tokenIn mismatch');
     require(input.amount == params.amountIn, 'amountIn mismatch');
     require(output.token == params.tokenOut, 'tokenOut mismatch');
     require(output.minAmountOut >= params.minAmountOut, 'minAmountOut
        mismatch');

     uint256 outBefore = IERC20(params.tokenOut).balanceOf(address(this));
     // ... continue with approvals and execution
     // (bool ok, bytes memory res) = params.to.call{value: params.value}(
        params.data);
     uint256 outAfter = IERC20(params.tokenOut).balanceOf(address(this));
     uint256 outDelta = outAfter > outBefore ? outAfter - outBefore : 0;
     require(outDelta >= params.minAmountOut, 'insufficient amountOut');
}
```

Team Response

Fixed.

[L-01] `StrategyManager` Inherits Pausable But Lacks Pause/Unpause Functionality

Severity

Low Risk

Description

The `StrategyManager` contract inherits from OpenZeppelin's `Pausable` contract, but does not expose any mechanism to trigger the pause or unpause states. Specifically, there are no external or public functions that call `_pause()` or `_unpause()`. Additionally, none of the contract's functions utilize the `whenNotPaused` or `whenPaused` modifiers.

This results in the `Pausable` inheritance being effectively dead code. It increases the contract's bytecode size and deployment cost without providing any functional benefit. It may also lead to confusion for developers or auditors who might assume the contract has an emergency stop mechanism due to the inheritance.

Location of Affected Code

File: contracts/StrategyManager.sol

```
contract StrategyManager is ReentrancyGuard, Pausable, AccessControl {
```

Impact

- **Unnecessary Gas Costs**: The inheritance increases the size of the deployed bytecode, leading to higher deployment costs.
- **Misleading Architecture**: The presence of `Pausable` in the inheritance list suggests that the contract supports an emergency pause functionality, which is not actually implemented or reachable.

Recommendation

If the pause functionality is intended: 1. Implement `pause()` and `unpause()` functions. 2. Restrict these functions to an appropriate role (e.g., `MANAGER_ROLE` or `DEFAULT_ADMIN_ROLE`). 3. Apply the `whenNotPaused` modifier to critical functions (e.g., `executeAllocSingle()`, `executeAllocPair()`, `withdrawSingleAll()`, `withdrawPairAll()`).

Example:

```
function pause() external onlyAdmin {
    _pause();
}

function unpause() external onlyAdmin {
    _unpause();
}

// Apply modifier to critical functions
function executeAllocSingle(bytes32 key) external onlyManager
   nonReentrant whenNotPaused {
    // code
}
```

Team Response

Fixed.

[L-02] Harvest Function Denial of Service Due to Single Strategy Revert

Severity

Low Risk

Description

The `harvest()` function in `StrategyManager` iterates through all registered single and pair allocations to harvest rewards. However, it performs these external calls to strategies without using a `try/catch` block.

If a single strategy reverts (e.g., due to a paused state, temporary failure, or bug in that specific strategy), the entire `harvest()` transaction will revert. This causes a Denial of Service (DoS) for the harvesting process of all other functioning strategies in the same batch.

While `harvestSingleAllocation()` and `harvestPairAllocation()` exist for individual harvesting, the batch `harvest()` function is intended for operational efficiency and automation. A single failure rendering the batch function unusable degrades this efficiency.

Location of Affected Code

File: contracts/StrategyManager.sol#L378-L394

```
function harvest() external onlyManager nonReentrant {
    for (uint256 i; i < singleKeys.length; ) {
        SingleAllocation storage alloc = singleAllocs[singleKeys[i]];
        if (alloc.exists) ISingleAssetStrategy(alloc.strategy).harvest(
           alloc.token);
        unchecked {
            ++i;
        }
    }

    for (uint256 i; i < pairKeys.length; ) {
        PairAllocation storage alloc = pairAllocs[pairKeys[i]];
        if (alloc.exists) IPairAssetStrategy(alloc.strategy).harvest(
           alloc.tokenA, alloc.tokenB);
        unchecked {
            ++i;
        }
    }
}
```

Impact

The automation of reward harvesting is disrupted. If one strategy fails, the manager or keeper cannot use the `harvest()` function to collect rewards from healthy strategies. They are forced to identify the failing strategy and then manually call individual harvest functions for every other strategy, increasing operational complexity and gas costs.

Proof of Concept

1. The `StrategyManager` has two strategies registered: Strategy A (working) and Strategy B (broken/reverting).
2. The manager calls `harvest()` to collect rewards from all strategies.
3. The call to Strategy A succeeds.
4. The call to Strategy B reverts.
5. The entire `harvest()` transaction reverts, and no rewards are collected from Strategy A.

Recommendation

Wrap the external `harvest()` calls in a `try/catch` block. This allows the function to continue processing other strategies even if one fails. Ideally, emit an event upon failure to alert off-chain monitoring systems.

```
function harvest() external onlyManager nonReentrant {
    for (uint256 i; i < singleKeys.length; ) {
        SingleAllocation storage alloc = singleAllocs[singleKeys[i]];
        if (alloc.exists) {
            try ISingleAssetStrategy(alloc.strategy).harvest(alloc.token)
                {
                // Success
            } catch (bytes memory reason) {
                emit HarvestFailed(singleKeys[i], reason);
            }
        }
        unchecked {
            ++i;
        }
    }
    // ... similar logic for pairKeys
}
```

Team Response

Fixed.

[L-03] Withdraw Priority Updates Lack Validation of Allocation Keys

Severity

Low Risk

Description

The `StrategyManager.setWithdrawPriority()` function updates the withdrawal queue without validating that the provided keys correspond to existing allocations. Although the function is restricted to the manager, the absence of on-chain checks increases the risk of misconfiguration, such as adding non-existent keys or accidentally omitting active strategies.

Location of Affected Code

File: contracts/StrategyManager.sol#L298-L306

```
function setWithdrawPriority(WithdrawPriority[] calldata queue) external
    onlyManager {
     delete withdrawPriority;
     for (uint256 i; i < queue.length; ) {
          withdrawPriority.push(queue[i]);
          unchecked {
              ++i;
          }
     }
}
```

Impact

If the manager inadvertently uploads a priority list with invalid keys or an empty list while relying on strategies for liquidity, user withdrawals could revert with `VAULT__INSUFFICIENT_FUNDS` until the configuration is corrected. This is an operational risk rather than a direct vulnerability.

Recommendation

It is recommended to add basic sanity checks to `setWithdrawPriority()` to ensure that every key in the provided queue corresponds to a valid, existing allocation.

```
function setWithdrawPriority(WithdrawPriority[] calldata queue) external
    onlyManager {
     delete withdrawPriority;
     for (uint256 i; i < queue.length; ) {
          // Validation: Ensure the allocation actually exists
          if (queue[i].kind == StrategyKind.SingleAsset) {
              if (!singleAllocs[queue[i].allocKey].exists) revert
                 VAULT__ALLOCATION_NOT_FOUND();
          } else {
              if (!pairAllocs[queue[i].allocKey].exists) revert
                 VAULT__ALLOCATION_NOT_FOUND();
          }

          withdrawPriority.push(queue[i]);
          unchecked {
              ++i;
          }
     }
}
```

Team Response

Fixed.

[I-01] Missing Zero Amount Validation in Deposit and Withdraw Functions

Severity

Informational Risk

Description

The `deposit()` and `withdraw()` functions in `PrimeStrategy.sol` do not validate that the `amount` parameter is greater than zero. This allows users to execute transactions with zero amounts, consuming gas unnecessarily, emitting misleading events, and potentially confusing off-chain systems.

Location of Affected Code

File: contracts/PrimeStrategy.sol

```
function deposit(address token, uint256 amount) external nonReentrant {
    // @audit missing zero amount validation
    IERC20(token).safeTransferFrom(msg.sender, address(this), amount);
    // code
}

function withdraw(address token, uint256 amount) external nonReentrant {
    // @audit missing zero amount validation
    // code
}
```

Impact

Users pay gas fees for zero-amount transactions that have no effect.

Proof of Concept

Call `deposit(USDC, 0)` or `withdraw(USDC, 0)`. The transaction succeeds, consumes gas, and emits events with zero amounts, providing no meaningful effect.

Recommendation

Add zero amount validation: `require(amount > 0, "Amount must be greater than zero");` at the beginning of both functions.

Team Response

Fixed.

[I-02] Missing Event Emissions for Administrative Parameter Updates

Severity

Informational Risk

Description

Administrative functions in `StrategyManager` (like `setWithdrawPriority()`, `setTreasury()`) and the `executeAllocSingle()` function do not emit events upon execution. This lack of event emission prevents off-chain monitoring tools and users from tracking important configuration changes and fund movements in real-time.

Location of Affected Code

File: contracts/StrategyManager.sol

Impact

Users and off-chain tools (like indexers or graphs) can't track the history of the vault correctly. If the manager changes the treasury address or reorders the withdrawal queue, nobody knows until they inspect the contract state manually.

Recommendation

Define and emit events for all state-changing functions. This makes it easy to track what the Manager is doing.

Add these events to `StrategyManager.sol` (or an interface):

```
event TreasurySet(address indexed newTreasury);
event WithdrawPrioritySet(WithdrawPriority[] priority);
event SingleAllocationSet(bytes32 indexed key, address indexed strategy,
   uint256 cap);
event PairAllocationSet(bytes32 indexed key, address indexed strategy,
   uint256 capA, uint256 capB);
event SingleAllocationExecuted(bytes32 indexed key, uint256 allocated);
```

And update the functions to emit them:

```
function setTreasury(address _treasury) external onlyAdmin {
    treasury = _treasury;
    emit TreasurySet(_treasury);
}

function executeAllocSingle(bytes32 key) external onlyManager
   nonReentrant {
    // ... execution logic ...
    emit SingleAllocationExecuted(key, alloc.allocated);
}
```

Team Response

Fixed.

[I-03] Missing Zero Address Checks

Severity

Informational Risk

Description

The contracts `StrategyManager.sol`, `PrimeStrategy.sol`, and `StrategyRegistry.sol` accept address parameters without verifying they are not the zero address (`address(0)`). Missing these checks can lead to misconfiguration, loss of privileges, or loss of funds.

Location of Affected Code

File: 'contracts/StrategyManager.sol

```
constructor(address _strategyRegistry, address _manager, address
    _treasury, address _admin) {
     // ...
}

function addSingleAllocation(address token, uint256 cap, address stra)
    external onlyManager {
     // ...
}

function addPairAllocation(address tokenA, address tokenB, uint256 capA,
    uint256 capB, address stra) external onlyManager {
     // code
}

function updateRole(address user, bytes32 role, bool grant) external
    onlyAdmin {
     // code
}
```

```
function setTreasury(address _treasury) external onlyAdmin {
    // code
}

function topUpIL(address token, uint256 amount) external onlyManager {
    // code
}

function swap(SwapParams calldata params) external onlyManager
   nonReentrant {
    // params.tokenIn, params.tokenOut, params.to checks missing
    // code
}
```

File: contracts/PrimeStrategy.sol

```
constructor(address _manager, address _vaultRegistry, address _treasury)
    StrategyManager(_vaultRegistry, _manager, _treasury, msg.sender)
{}

function deposit(address token, uint256 amount) external nonReentrant {
    // code
}

function withdraw(address token, uint256 amount) external nonReentrant {
    // code
}
```

File: contracts/StrategyRegistry.sol#L53-L76

```
function addStrategy(address stra, address expectedVault) external
   onlyOwner {
  // code
}
```

Impact

Missing zero address checks can lead to severe consequences. Deploying contracts with zero addresses as initial parameters forces a costly redeployment. Setting critical roles like admin or manager to the zero address permanently locks privileges, making the contract unmanageable. Furthermore, configuring the treasury to the zero address results in irreversible loss of funds (burning), while allocating zero address tokens or strategies causes operational failures, reverts, and potential asset loss.

Recommendation

Add checks to ensure that critical address parameters are not zero.

```
require(_address != address(0), "Invalid Address");
```

Team Response

Fixed.

shieldify

Thank you!