BLOCKSEC

Security Audit

Report for dstock-core-evm

Date: November 18, 2025 **Version:** 1.0

Contact: contact@blocksec.com

Contents

Report Manifest

Item	Description
Client	DStock
Target	dstock-core-evm

Version History

Version	Date	Description
1.0	November 18, 2025	First release

Signature

Digitally signed by: BlockSec (https://blocksec.com)

SHA1 digest of public key: BAB106D3708BC6DE0963CE05E9478C24B326AE1C

Date: 2025-11-18 09:23:11+0000

About BlockSec BlockSec focuses on the security of the blockchain ecosystem and collaborates with leading DeFi projects to secure their products. BlockSec is founded by top-notch security researchers and experienced experts from both academia and industry. They have published multiple blockchain security papers in prestigious conferences, reported several zero-day attacks of DeFi applications, and successfully protected digital assets that are worth more than 14 million dollars by blocking multiple attacks. They can be reached at Email, Twitter and Medium.

Chapter 1 Introduction

1.1 About Target Contracts

Information	Description
Type	Smart Contract
Language	Solidity
Approach	Semi-automatic and manual verification

The target of this audit is the code repository [1] of dstock-core-evm of DStock.

The DStock is a Real-World Asset (RWA) token aggregation system that is upgradable through the Beacon proxy pattern. The project consists of three core smart contracts and they are `DStockWrapper`, `DStockFactoryRegistry`, and `DStockCompliance`.

The contract `DStockWrapper` is an ERC-20 upgradeable token representing unified DStock shares. The contract `DStockFactoryRegistry` manages multiple wrapper instances using the Beacon pattern, while `DStockCompliance` enforces user-level compliance checks for each supported RWA token. To obtain DStock shares, users must wrap their underlying RWA tokens via the `DStockWrapper` contract, after which the system mints corresponding DStock shares for them. For redemption, users must unwrap their DStock shares via the contract `DStockWrapper` to receive any supported underlying RWA token. In addition, the project's admin role can manage supported RWA tokens and compliance rules through `DStockCompliance`, while the wrapper logic upgrade is managed via the Beacon proxy pattern in `DStockFactoryRegistry`.

Note this audit only focuses on the smart contracts in the following directories/files:

- src

Other files are not within the scope of the audit. Additionally, all dependencies of the smart contracts within the audit scope are considered reliable in terms of both functionality and security, and are therefore not included in the audit scope.

The auditing process is iterative. Specifically, we would audit the commits that fix the discovered issues. If there are new issues, we will continue this process. The commit SHA values during the audit are shown in the following table. Our audit report is responsible for the code in the initial version (`Version 1`), as well as new code (in the following versions) to fix issues in the audit report. Code prior to and including the baseline version (`Version 0`), where applicable, is outside the scope of this audit and assumes to be reliable and secure.

Project	Version	Commit Hash
dstock-core-evm	`Version 1`	`211ba90f9b3fc5c8a1fd7eb05052062925ccc969`
	`Version 2`	`1fa91e03cb93219d7f29d85ea37dc352a5eb3a5b`

[1] https://github.com/dstockofficial/dstock-core-evm

1.2 Disclaimer

This audit report does not constitute investment advice or a personal recommendation. It does not consider, and should not be interpreted as considering or having any bearing on, the potential economics of a token, token sale or any other product, service or other asset. Any entity should not rely on this report in any way, including for the purpose of making any decisions to buy or sell any token, product, service or other asset.

This audit report is not an endorsement of any particular project or team, and the report does not guarantee the security of any particular project. This audit does not give any warranties on discovering all security issues of the smart contracts, i.e., the evaluation result does not guarantee the nonexistence of any further findings of security issues. As one audit cannot be considered comprehensive, we always recommend proceeding with independent audits and a public bug bounty program to ensure the security of smart contracts.

The scope of this audit is limited to the code mentioned in Section 1.1. Unless explicitly specified, the security of the language itself (e.g., the solidity language), the underlying compiling toolchain and the computing infrastructure are out of the scope.

1.3 Procedure of Auditing

We perform the audit according to the following procedure.

- **Vulnerability Detection** We first scan smart contracts with automatic code analyzers, and then manually verify (reject or confirm) the issues reported by them.
- **Semantic Analysis** We study the business logic of smart contracts and conduct further investigation on the possible vulnerabilities using an automatic fuzzing tool (developed by our research team). We also manually analyze possible attack scenarios with independent auditors to cross-check the result.
- **Recommendation** We provide some useful advice to developers from the perspective of good programming practice, including gas optimization, code style, and etc.

We show the main concrete checkpoints in the following.

1.3.1 Security Issues

* Access control
* Permission management
* Whitelist and blacklist mechanisms
* Initialization consistency
* Improper use of the proxy system
* Reentrancy
* Denial of Service (DoS)
* Untrusted external call and control flow
* Exception handling
* Data handling and flow
* Events operation
* Error-prone randomness

- Oracle security
- Business logic correctness
- Semantic and functional consistency
- Emergency mechanism
- Economic and incentive impact

1.3.2 Additional Recommendation

- Gas optimization
- Code quality and style

Note *The previous checkpoints are the main ones. We may use more checkpoints during the auditing process according to the functionality of the project.*

1.4 Security Model

To evaluate the risk, we follow the standards or suggestions that are widely adopted by both industry and academy, including OWASP Risk Rating Methodology [2] and Common Weakness Enumeration [3]. The overall *severity* of the risk is determined by *likelihood* and *impact*. Specifically, likelihood is used to estimate how likely a particular vulnerability can be uncovered and exploited by an attacker, while impact is used to measure the consequences of a successful exploit.

In this report, both likelihood and impact are categorized into two ratings, i.e., *high* and *low* respectively, and their combinations are shown in Table 1.1.

Table 1.1: Vulnerability Severity Classification

Impact \ Likelihood	*High*	*Low*
High	High	Medium
Low	Medium	Low

Accordingly, the severity measured in this report are classified into three categories: **High**, **Medium**, **Low**. For the sake of completeness, **Undetermined** is also used to cover circumstances when the risk cannot be well determined.

Furthermore, the status of a discovered item will fall into one of the following five categories:

- **Undetermined** No response yet.
- **Acknowledged** The item has been received by the client, but not confirmed yet.

[2] https://owasp.org/www-community/OWASP_Risk_Rating_Methodology

[3] https://cwe.mitre.org/

- **Confirmed** The item has been recognized by the client, but not fixed yet.
- **Partially Fixed** The item has been confirmed and partially fixed by the client.
- **Fixed** The item has been confirmed and fixed by the client.

Chapter 2 Findings

In total, we found **fourteen** potential security issues. Besides, we have **five** recommendations and **five** notes.

- High Risk: 3
- Medium Risk: 6
- Low Risk: 5
- Recommendation: 5
- Note: 5

ID	Severity	Description	Category	Status
1	High	Invalid pause mechanism integration in the contract `DStockWrapper`	Security Issue	Fixed
2	High	Accounting incompatibility with rebase tokens	Security Issue	Fixed
3	High	Lack of liquidity update during stock splitting	Security Issue	Fixed
4	Medium	Incorrect accounting in the function `unwrap()`	Security Issue	Fixed
5	Medium	Potential DoS issue for functions `addUnderlyings()` and `migrateUnderlyings()`	Security Issue	Fixed
6	Medium	Inaccurate returned values in functions `totalSupply()` and `balanceOf()`	Security Issue	Fixed
7	Medium	Precision loss in the function `unwrap()`	Security Issue	Fixed
8	Medium	Fees remain locked in the contract `DStockWrapper` when `treasury` is not set	Security Issue	Fixed
9	Medium	Lack of a mechanism to collect holding fees	Security Issue	Fixed
10	Low	Unable to add underlying tokens after removal or migration	Security Issue	Fixed
11	Low	Incorrect check on `liquidToken` in function `unwrap()`	Security Issue	Fixed
12	Low	Potential circumvention of factory registration validation	Security Issue	Fixed
13	Low	Lack of deprecated wrapper validation	Security Issue	Fixed
14	Low	Inconsistent underlying asset removal	Security Issue	Fixed
15	-	Rename return parameter in the function `underlyingInfo()`	Recommendation	Fixed
16	-	Remove redundant code in the contract `DStockWrapper`	Recommendation	Fixed
17	-	Validate new values in setter functions	Recommendation	Fixed

18	-	Lack of invoking function `_disableInitializers()`	Recommendation	Fixed
19	-	Implement logic for old wrapper after migration	Recommendation	Fixed
20	-	Potential centralization risks	Note	-
21	-	Assumption of underlying token value equivalence	Note	-
22	-	Potential risks regarding peg-out events	Note	-
23	-	Ensure consistent fee rates among assets in the same wrapper	Note	-
24	-	Time gaps in split and merge adjustments	Note	-

The details are provided in the following sections.

2.1 Security Issue

2.1.1 Invalid pause mechanism integration in the contract `DStockWrapper`

Severity High

Status Fixed in `Version 2`

Introduced by `Version 1`

Description The contract `DStockWrapper` implements a pause mechanism using both Openzeppelin's `PausableUpgradeable` and the state variable `pausedByFactory`. However, the integration between these two systems is incomplete. The contract uses the modifier `whenNotPaused` for access control, which only checks the native OpenZeppelin pause state. This modifier does not account for the custom `pausedByFactory` variable. Consequently, the factory-initiated pause state cannot effectively suspend contract operations.

```
310 function setPausedByFactory(bool p) external {
311   if (msg.sender != factoryRegistry && !hasRole(PAUSER_ROLE, msg.sender)) revert NotAllowed();
312   pausedByFactory = p;
313 }
```

Listing 2.1: src/DStockWrapper.sol

```
151 function wrap(address token, uint256 amount, address to)
152   external
153   nonReentrant
154   whenNotPaused
155   updateMultiplier
156   returns (uint256 net18, uint256 mintedShares)
```

Listing 2.2: src/DStockWrapper.sol

```
192 function unwrap(address token, uint256 amount, address to)
193   external
```

```
194    nonReentrant
195    whenNotPaused
196    updateMultiplier
```

Listing 2.3: src/DStockWrapper.sol

```
233 function _update(address from, address to, uint256 value)
234    internal
235    override
236    updateMultiplier
237    whenNotPaused
```

Listing 2.4: src/DStockWrapper.sol

Impact This design flaw makes the factory pause mechanism ineffective.

Suggestion Modify the `whenNotPaused` modifier or create a new custom modifier that checks both the OpenZeppelin pause state and the `pausedByFactory` variable.

2.1.2 Accounting incompatibility with rebase tokens

Severity High

Status Fixed in `Version 2`

Introduced by `Version 1`

Description The annotations indicate that the contract `DStockWrapper` uses an underlying token similar to `TSLAx`, which implements rebase mechanisms that automatically adjust token balances according to the TSLAx contract implementation on the EVM compatiable chains (e.g., BSC).

```
17/// @title DStockWrapper (multi-underlying, safe decimals rescaling)
18/// @notice Multiple underlyings (e.g. TSLAx/TSLAy/...) can map to ONE d-stock.
19///         Shares x multiplier (Ray=1e18) accounting; BPS fees; holding fee; split; force move.
20///         This version adds safe rescaling for arbitrary token decimals (both <18 and >18).
```

Listing 2.5: src/DStockWrapper.sol

The project tracks redeemable liquidity using the state variable `liquidToken`, which records the underlying token balance. However, this approach is incompatible with underlying tokens being rebase tokens, as it fails to account for their dynamic balance adjustments. This flaw may result in several issues as follows:

1. When the rebase token deflates (negative rebase), the contract's holding balance decreases and is less than the `liquidToken`. This discrepancy causes transfer operations to fail, resulting in a DoS issue for the function `unwrap()`.

2. When the rebase token inflates (positive rebase), the contract's holding balance increases and is more than the `liquidToken`. The liquidity validation mechanism prevents redemption of these excess tokens, locking additional funds in the contract.

3. Standard token transfers through `safeTransferFrom()` can yield inconsistent amounts because of concurrent rebase adjustments.

```
151 function wrap(address token, uint256 amount, address to)
152   external
153   nonReentrant
154   whenNotPaused
155   updateMultiplier
156   returns (uint256 net18, uint256 mintedShares)
157 {
158   UnderlyingInfo storage info = underlyings[token];
159   if (info.decimals == 0) revert UnknownUnderlying();
160   if (!info.enabled)    revert UnsupportedUnderlying();
161
162   _checkCompliance(msg.sender, to, amount, 1 /* Wrap */);
163
164   IERC20(token).safeTransferFrom(msg.sender, address(this), amount);
165
166   uint256 gross18 = _normalize(token, amount);
167   uint256 fee18  = (wrapFeeBps == 0) ? 0 : (gross18 * wrapFeeBps) / 10_000;
168   net18 = gross18 - fee18;
169
170   // move fee to treasury (token units) and track net liquidity
171   if (fee18 > 0 && treasury != address(0)) {
172     uint256 feeToken = _denormalize(token, fee18);
173     if (feeToken > 0) IERC20(token).safeTransfer(treasury, feeToken);
174     info.liquidToken += (amount - feeToken);
175   } else {
176     info.liquidToken += amount;
177   }
```

Listing 2.6: src/DStockWrapper.sol

Impact Rebase token incompatibility causes DoS conditions, fund lockups, and other unpredictable behavior.

Suggestion Revise the code logic accordingly.

2.1.3 Lack of liquidity update during stock splitting

Severity High

Status Fixed in `Version 2`

Introduced by `Version 1`

Description In contract `DStockWrapper`, the function `applySplit()` updates the `multiplier` when there is a split or merge, which increases or decreases stock shares proportionally. However, the state `liquidToken` is not updated accordingly. For example, if the multiplier is set to a larger value, users can unwrap more tokens with the same shares. However, the maximum amount for unwrapping, i.e., `liquidToken`, remains unchanged. This can cause user shares to be locked in the contract, unable to withdraw fully.

```
315 function applySplit(uint256 numerator, uint256 denominator)
316   external
317   onlyRole(OPERATOR_ROLE)
```

```
318   updateMultiplier
319 {
320   require(numerator > 0 && denominator > 0, "bad ratio");
321   uint256 oldM = multiplier;
322   uint256 newM = (oldM * numerator) / denominator;
323   multiplier = newM > 0 ? newM : 1;
324   emit SplitApplied(numerator, denominator, oldM, multiplier);
325   emit MultiplierUpdated(multiplier);
326 }
```

Listing 2.7: src/DStockWrapper.sol

Impact User shares cannot fully withdraw from contract `DStockWrapper` when the multiplier is increased.

Suggestion Revise the logic accordingly.

2.1.4 Incorrect accounting in the function `unwrap()`

Severity Medium

Status Fixed in `Version 2`

Introduced by `Version 1`

Description The contract `DStockWrapper` uses `liquidToken` to track redeemable underlying amount, and functions `wrap()` and `unwrap()` will update this variable. However, the function `unwrap()` contains flawed logic. It deducts only the `netToken` amount from the `liquidToken` while actually transferring out both `netToken` and `feeToken`. This creates an accounting discrepancy where the recorded liquidity exceeds the contract's actual token balance.

```
192 function unwrap(address token, uint256 amount, address to)
193   external
194   nonReentrant
195   whenNotPaused
196   updateMultiplier
197 {
198   UnderlyingInfo storage info = underlyings[token];
199   if (info.decimals == 0) revert UnknownUnderlying();
200   if (!info.enabled)    revert UnsupportedUnderlying();
201
202   _checkCompliance(msg.sender, to, amount, 2 /* Unwrap */);
203
204   uint256 gross18 = _normalize(token, amount);
205
206   // burn shares (ceil to cover gross18 at current multiplier)
207   uint256 s = _toShares(gross18);
208   if (_toAmount(s) < gross18) s += 1;
209   if (_shares[msg.sender] < s) revert InsufficientShares();
210
211   uint256 fee18 = (unwrapFeeBps == 0) ? 0 : (gross18 * unwrapFeeBps) / 10_000;
212   uint256 net18 = gross18 - fee18;
213
214   uint256 feeToken = _denormalize(token, fee18);
```

```
215   uint256 netToken = _denormalize(token, net18);
216
217   if (info.liquidToken < netToken) revert InsufficientLiquidity();
218
219   _shares[msg.sender] -= s;
220   _totalShares      -= s;
221
222   info.liquidToken  -= netToken;
```

Listing 2.8: src/DStockWrapper.sol

Impact This accounting error results in an overstatement of available liquidity.

Suggestion Revise the code accordingly.

2.1.5 Potential DoS issue for functions `addUnderlyings()` and `migrateUnderlyings()`

Severity Medium

Status Fixed in `Version 2`

Introduced by `Version 1`

Description In the contract `DStockFactoryRegistry`, both the functions `addUnderlyings()`and `migrateUnderlyings()` invoke `IDStockWrapper(wrapper).addUnderlying(tokens[i])` to update wrapper state. However, the contract `DStockWrapper` restricts the function `addUnderlying()` to `OPERATOR_ROLE`. When the `DStockFactoryRegistry` contract is not granted this role, it can not successfully execute wrapper updates, resulting in a potential DoS (denial-of-service) issue.

```
164 function addUnderlyings(address wrapper, address[] calldata tokens)
165   external
166   onlyRole(OPERATOR_ROLE)
167 {
168   if (!isWrapper[wrapper]) revert NotRegistered();
169   if (tokens.length == 0) revert InvalidParams("empty tokens");
170
171   // Validate and ensure no conflicts before mutating state
172   for (uint256 i = 0; i < tokens.length; i++) {
173     address u = tokens[i];
174     if (u == address(0)) revert ZeroAddress();
175     if (wrapperOf[u] != address(0)) revert AlreadyRegistered();
176   }
177
178   // Call wrapper to enable each underlying, then map
179   for (uint256 i = 0; i < tokens.length; i++) {
180     IDStockWrapper(wrapper).addUnderlying(tokens[i]);
181     wrapperOf[tokens[i]] = wrapper;
182     emit UnderlyingMapped(tokens[i], wrapper);
183   }
184
185   emit UnderlyingsAdded(wrapper, tokens);
186 }
```

Listing 2.9: src/DStockFactoryRegistry.sol

```
409 function addUnderlying(address token) external onlyRole(OPERATOR_ROLE) {
410   _addUnderlying(token);
411 }
```

Listing 2.10: src/DStockWrapper.sol

```
199 function migrateUnderlyings(address[] calldata underlyings, address oldWrapper, address
        newWrapper)
200   external
201   onlyRole(DEFAULT_ADMIN_ROLE)
202 {
203   if (newWrapper == address(0) || oldWrapper == address(0)) revert ZeroAddress();
204   if (!isWrapper[oldWrapper]) revert InvalidParams("oldWrapper not a wrapper");
205   if (!isWrapper[newWrapper]) revert InvalidParams("newWrapper not a wrapper");
206   if (underlyings.length == 0) revert InvalidParams("empty underlyings");
207
208   // Validate ownership & conflicts
209   for (uint256 i = 0; i < underlyings.length; i++) {
210     address u = underlyings[i];
211     if (wrapperOf[u] != oldWrapper) revert InvalidParams("token not owned by oldWrapper");
212   }
213
214   // Add to new, remap
215   for (uint256 i = 0; i < underlyings.length; i++) {
216     address u = underlyings[i];
217     IDStockWrapper(newWrapper).addUnderlying(u);
218     wrapperOf[u] = newWrapper;
219     emit UnderlyingMapped(u, newWrapper);
220   }
221
222   emit UnderlyingsMigrated(oldWrapper, newWrapper, underlyings);
223 }
```

Listing 2.11: src/DStockFactoryRegistry.sol

Impact This design results in a DoS issue for adding underlying tokens through `DStockFactoryRegistry`.

Suggestion Grant the `OPERATOR_ROLE` to contract `DStockFactoryRegistry` within each `DStockWrapper`.

2.1.6 Inaccurate returned values in functions `totalSupply()` and `balanceOf()`

Severity Medium

Status Fixed in `Version 2`

Introduced by `Version 1`

Description In the contract `DStockWrapper`, functions `totalSupply()` and `balanceOf()` rely on the function `_toAmount()` for their return values. The function `_toAmount()` calculates the amount based on the `multiplier`. However, this variable only updates in the `updateMultiplier()` function. The `multiplier` becomes stale when the time since the last update exceeds a whole period. Consequently, these functions return values based on outdated data rather than the actual state.

```
146 function totalSupply() public view override returns (uint256) { return _toAmount(_totalShares);
        }
147 function balanceOf(address a) public view override returns (uint256) { return _toAmount(_shares[
        a]); }
```

Listing 2.12: src/DStockWrapper.sol

```
432 function _toAmount(uint256 shares) internal view returns (uint256) {
433   return (shares == 0) ? 0 : (shares * multiplier) / RAY;
434 }
```

Listing 2.13: src/DStockWrapper.sol

Impact This flaw may cause functions `totalSupply()` and `balanceOf()` to return inaccurate values.

Suggestion Modify these functions to calculate amounts using a freshly computed multiplier.

2.1.7 Precision loss in the function `unwrap()`

Severity Medium

Status Fixed in `Version 2`

Introduced by `Version 1`

Description In the contract `DStockWrapper`, the function `unwrap()` enables users to withdraw the underlying tokens. This function uses `gross18` (i.e., the normalized input `amount`) to calculate `fee18` and `net18`, then converts the normalized amounts back to the original token precision (i.e., `feeToken` and `netToken`) for the transfer. Considering the precision loss due to denormalization, the sum of `feeToken` and `netToken` may be less than the user's input amount.

Additionally, when processing small amounts, particularly for tokens with fewer than 18 decimals, the calculated `netToken` may round down to zero. For example, unwrapping 1 wei of a 6-decimal token with a 1 basis point fee results in a `netToken` of `0`. Despite receiving zero tokens, the user's shares are still burned.

```
192 function unwrap(address token, uint256 amount, address to)
193   external
194   nonReentrant
195   whenNotPaused
196   updateMultiplier
197 {
198   UnderlyingInfo storage info = underlyings[token];
199   if (info.decimals == 0) revert UnknownUnderlying();
200   if (!info.enabled)    revert UnsupportedUnderlying();
201
202   _checkCompliance(msg.sender, to, amount, 2 /* Unwrap */);
203
204   uint256 gross18 = _normalize(token, amount);
205
206   // burn shares (ceil to cover gross18 at current multiplier)
207   uint256 s = _toShares(gross18);
208   if (_toAmount(s) < gross18) s += 1;
```

```
209   if (_shares[msg.sender] < s) revert InsufficientShares();
210
211   uint256 fee18 = (unwrapFeeBps == 0) ? 0 : (gross18 * unwrapFeeBps) / 10_000;
212   uint256 net18 = gross18 - fee18;
213
214   uint256 feeToken = _denormalize(token, fee18);
215   uint256 netToken = _denormalize(token, net18);
216
217   if (info.liquidToken < netToken) revert InsufficientLiquidity();
218
219   _shares[msg.sender] -= s;
220   _totalShares      -= s;
221
222   info.liquidToken  -= netToken;
223   if (feeToken > 0 && treasury != address(0)) {
224     IERC20(token).safeTransfer(treasury, feeToken);
225   }
226   IERC20(token).safeTransfer(to, netToken);
227
228   emit Unwrapped(token, msg.sender, to, gross18, fee18, net18, s);
229   emit Transfer(msg.sender, address(0), _toAmount(s));
230 }
```

Listing 2.14: src/DStockWrapper.sol

Impact This flaw results in the loss of user shares without any token compensation.

Suggestion Deduct the `feeToken` from the `amount` to obtain the correct `netToken` value.

2.1.8 Fees remain locked in the contract `DStockWrapper` when `treasury` is not set

Severity Medium

Status Fixed in `Version 2`

Introduced by `Version 1`

Description In function `wrap()`, when `wrapFeeBps` is greater than zero but `treasury` is not set, calculated fees are deducted from user deposits. However, fees remain locked in the contract since shares are minted only based on the `net18`. Meanwhile, the same issue occurs in the function `unwrap()`, where fees remain locked in the contract when `treasury` is not set.

These accrued wrap and unwrap fees have no extraction mechanism, which leads to fees being locked in the contract. In both cases, the only recovery method is to update the `multiplier` through the function `applySplit()`.

```
151 function wrap(address token, uint256 amount, address to)
152   external
153   nonReentrant
154   whenNotPaused
155   updateMultiplier
156   returns (uint256 net18, uint256 mintedShares)
157 {
158   UnderlyingInfo storage info = underlyings[token];
159   if (info.decimals == 0) revert UnknownUnderlying();
```

```
160   if (!info.enabled)    revert UnsupportedUnderlying();
161
162   _checkCompliance(msg.sender, to, amount, 1 /* Wrap */);
163
164   IERC20(token).safeTransferFrom(msg.sender, address(this), amount);
165
166   uint256 gross18 = _normalize(token, amount);
167   uint256 fee18  = (wrapFeeBps == 0) ? 0 : (gross18 * wrapFeeBps) / 10_000;
168   net18 = gross18 - fee18;
169
170   // move fee to treasury (token units) and track net liquidity
171   if (fee18 > 0 && treasury != address(0)) {
172     uint256 feeToken = _denormalize(token, fee18);
173     if (feeToken > 0) IERC20(token).safeTransfer(treasury, feeToken);
174     info.liquidToken += (amount - feeToken);
175   } else {
176     info.liquidToken += amount;
177   }
178
179   uint256 s = _toShares(net18);
180   if (s == 0) revert TooSmall();
181   if (cap != 0 && _toAmount(_totalShares + s) > cap) revert CapExceeded();
182
183   _shares[to] += s;
184   _totalShares += s;
185   mintedShares = s;
186
187   emit Wrapped(token, msg.sender, to, gross18, fee18, net18, s);
188   emit Transfer(address(0), to, _toAmount(s));
189 }
```

Listing 2.15: src/DStockWrapper.sol

```
223   if (feeToken > 0 && treasury != address(0)) {
224     IERC20(token).safeTransfer(treasury, feeToken);
225   }
```

Listing 2.16: src/DStockWrapper.sol

Impact Accumulated wrap and unwrap fees cannot be directly withdrawn from the wrapper when the `treasury` is not set.

Suggestion Implement a fee withdrawal method to extract accumulated fees.

2.1.9 Lack of a mechanism to collect holding fees

Severity Medium

Status Fixed in `Version 2`

Introduced by `Version 1`

Description In contract `DStockWrapper`, function `_applyAccruedFee()` updates the `multiplier` to passively collect holding fees, and privileged functions allow configuration of wrap and

unwrap fees. However, the contract lacks an external function to transfer the underlying tokens that represent the value accrued via the `multiplier` reduction.

```
421 modifier updateMultiplier() {
422   _applyAccruedFee();
423   _;
424 }
```

Listing 2.17: src/DStockWrapper.sol

```
494 // lazy holding-fee: m *= (1 - f)^n
495 function _applyAccruedFee() internal {
496   (uint256 m, uint256 periods) = _previewApplyAccruedFee(multiplier, lastTimeFeeApplied);
497   if (periods == 0) return;
498   multiplier = m > 0 ? m : 1;
499   lastTimeFeeApplied = uint64(block.timestamp - ((block.timestamp - lastTimeFeeApplied) %
          periodLength));
500   emit MultiplierUpdated(multiplier);
501 }
```

Listing 2.18: src/DStockWrapper.sol

Impact Accrued holding fees are locked in the contract, resulting in loss of protocol revenue.

Suggestion Revise the code logic accordingly.

2.1.10 Unable to add underlying tokens after removal or migration

Severity Low

Status Fixed in `Version 2`

Introduced by `Version 1`

Description The contract has an inconsistent state management issue between contracts `DStockFactoryRegistry` and `DStockWrapper` when removing or migrating underlying tokens. When the function `removeUnderlyingMapping()` or `migrateUnderlyings()` is invoked in the factory, it only clears the `wrapperOf[underlying]` but does not clear the underlying's state in the wrapper itself. The wrapper still retains the token's `decimals` in its `underlyings` mapping.

Subsequently, when re-adding the same underlying token to the same wrapper through `addUnderlyings()`, the factory invokes `wrapper.addUnderlying()`, which internally invokes function `_addUnderlying()`. This function checks if the token already exists by verifying `(info.decimals != 0)`. Since the `decimals` were not cleared from the wrapper's storage, the check fails, blocking the registration attempt.

```
190 function removeUnderlyingMapping(address underlying) external onlyRole(OPERATOR_ROLE) {
191   address w = wrapperOf[underlying];
192   if (w == address(0)) revert NotRegistered();
193   wrapperOf[underlying] = address(0);
194   emit UnderlyingUnmapped(underlying, w);
195 }
```

Listing 2.19: src/DStockFactoryRegistry.sol

```
525 function _addUnderlying(address token) internal {
526   if (token == address(0)) revert ZeroAddress();
527   UnderlyingInfo storage info = underlyings[token];
528   if (info.decimals != 0) revert NotAllowed(); // already exists
529   uint8 dec = IERC20Metadata(token).decimals(); // no limit now; we rescale safely both ways
530   info.decimals  = dec;
531   info.enabled   = true;
532   info.liquidToken = 0;
533   allUnderlyings.push(token);
534   emit UnderlyingAdded(token, dec);
535 }
```

Listing 2.20: src/DStockWrapper.sol

Impact Previously removed or migrated underlying tokens become unregistrable to the same wrapper contract.

Suggestion Revise the logic accordingly.

2.1.11 Incorrect check on `liquidToken` in function `unwrap()`

Severity Low

Status Fixed in `Version 2`

Introduced by `Version 1`

Description The function `unwrap()` charges an unwrap fee when the fee rate and the `treasury` are configured, and the fees are directly transferred to the `treasury` (line 224). However, the check on line 217 is incorrect in such cases. Specifically, the check does not ensure `liquidToken` can cover the total transferred amount, i.e., `netToken` and `feeToken`.

```
191 /// @notice Unwrap `amount` (token units) of `token` out to `to`.
192 function unwrap(address token, uint256 amount, address to)
193   external
194   nonReentrant
195   whenNotPaused
196   updateMultiplier
197 {
198   UnderlyingInfo storage info = underlyings[token];
199   if (info.decimals == 0) revert UnknownUnderlying();
200   if (!info.enabled)    revert UnsupportedUnderlying();
201
202   _checkCompliance(msg.sender, to, amount, 2 /* Unwrap */);
203
204   uint256 gross18 = _normalize(token, amount);
205
206   // burn shares (ceil to cover gross18 at current multiplier)
207   uint256 s = _toShares(gross18);
208   if (_toAmount(s) < gross18) s += 1;
209   if (_shares[msg.sender] < s) revert InsufficientShares();
210
211   uint256 fee18 = (unwrapFeeBps == 0) ? 0 : (gross18 * unwrapFeeBps) / 10_000;
212   uint256 net18 = gross18 - fee18;
```

```
213
214   uint256 feeToken = _denormalize(token, fee18);
215   uint256 netToken = _denormalize(token, net18);
216
217   if (info.liquidToken < netToken) revert InsufficientLiquidity();
218
219   _shares[msg.sender] -= s;
220   _totalShares      -= s;
221
222   info.liquidToken  -= netToken;
223   if (feeToken > 0 && treasury != address(0)) {
224     IERC20(token).safeTransfer(treasury, feeToken);
225   }
226   IERC20(token).safeTransfer(to, netToken);
227
228   emit Unwrapped(token, msg.sender, to, gross18, fee18, net18, s);
229   emit Transfer(msg.sender, address(0), _toAmount(s));
230 }
```

Listing 2.21: src/DStockWrapper.sol

Impact The check on liquidity could pass, but the function `unwrap()` execution will fail when the underlying liquidity is insufficient.

Suggestion Revise the check on `liquidToken`.

2.1.12 Potential circumvention of factory registration validation

Severity Low

Status Fixed in `Version 2`

Introduced by `Version 1`

Description Contract `DStockFactoryRegistry` is designed to be the central point for registering underlying assets and enforces that one asset be registered in only one wrapper contract (line 175). However, this restriction can be circumvented in contract `DStockWrapper`. Specifically, its function `addUnderlying()` can be invoked by `OPERATOR_ROLE`. Since the factory is not the sole possessor of the `OPERATOR_ROLE`, a privileged account can directly invoke the function `addUnderlying()`, circumventing the factory's check.

```
164 function addUnderlyings(address wrapper, address[] calldata tokens)
165   external
166   onlyRole(OPERATOR_ROLE)
167 {
168   if (!isWrapper[wrapper]) revert NotRegistered();
169   if (tokens.length == 0) revert InvalidParams("empty tokens");
170
171   // Validate and ensure no conflicts before mutating state
172   for (uint256 i = 0; i < tokens.length; i++) {
173     address u = tokens[i];
174     if (u == address(0)) revert ZeroAddress();
175     if (wrapperOf[u] != address(0)) revert AlreadyRegistered();
176   }
```

```
177
178   // Call wrapper to enable each underlying, then map
179   for (uint256 i = 0; i < tokens.length; i++) {
180     IDStockWrapper(wrapper).addUnderlying(tokens[i]);
181     wrapperOf[tokens[i]] = wrapper;
182     emit UnderlyingMapped(tokens[i], wrapper);
183   }
184
185   emit UnderlyingsAdded(wrapper, tokens);
186 }
```

Listing 2.22: src/DStockFactoryRegistry.sol

```
409 function addUnderlying(address token) external onlyRole(OPERATOR_ROLE) {
410   _addUnderlying(token);
411 }
```

Listing 2.23: src/DStockWrapper.sol

Impact Underlying assets can be added to wrappers without going through the factory's registration process and validation.

Suggestion Implement a check in contract `DStockWrapper`'s function `addUnderlying()`.

Clarification from BlockSec The issue is fixed by restricting function `addUnderlying()` to the factory, and the factory now supports mappings from one underlying token to a list of wrappers.

2.1.13 Lack of deprecated wrapper validation

Severity Low

Status Fixed in `Version 2`

Introduced by `Version 1`

Description The contract `DStockFactoryRegistry` implements the function `deprecate()` to mark a wrapper as deprecated. However, functions `migrateUnderlyings()` and `addUnderlyings()` lack validation for deprecated wrappers. This omission allows tokens to be added to or migrated into a wrapper that is designated as obsolete.

```
164 function addUnderlyings(address wrapper, address[] calldata tokens)
165   external
166   onlyRole(OPERATOR_ROLE)
167 {
168   if (!isWrapper[wrapper]) revert NotRegistered();
169   if (tokens.length == 0) revert InvalidParams("empty tokens");
```

Listing 2.24: src/DStockFactoryRegistry.sol

```
199 function migrateUnderlyings(address[] calldata underlyings, address oldWrapper, address
        newWrapper)
200   external
201   onlyRole(DEFAULT_ADMIN_ROLE)
202 {
203   if (newWrapper == address(0) || oldWrapper == address(0)) revert ZeroAddress();
```

```
204   if (!isWrapper[oldWrapper]) revert InvalidParams("oldWrapper not a wrapper");
205   if (!isWrapper[newWrapper]) revert InvalidParams("newWrapper not a wrapper");
206   if (underlyings.length == 0) revert InvalidParams("empty underlyings");
207
208   // Validate ownership & conflicts
209   for (uint256 i = 0; i < underlyings.length; i++) {
210     address u = underlyings[i];
211     if (wrapperOf[u] != oldWrapper) revert InvalidParams("token not owned by oldWrapper");
212   }
213
214   // Add to new, remap
215   for (uint256 i = 0; i < underlyings.length; i++) {
216     address u = underlyings[i];
217     IDStockWrapper(newWrapper).addUnderlying(u);
218     wrapperOf[u] = newWrapper;
219     emit UnderlyingMapped(u, newWrapper);
220   }
221
222   emit UnderlyingsMigrated(oldWrapper, newWrapper, underlyings);
223 }
```

Listing 2.25: src/DStockFactoryRegistry.sol

Impact A deprecated wrapper's continued modification creates state inconsistencies and risks directing users to an unsupported contract.

Suggestion Integrate deprecated wrapper checks into these functions.

2.1.14 Inconsistent underlying asset removal

Severity Low

Status Fixed in `Version 2`

Introduced by `Version 1`

Description In contract `DStockFactoryRegistry`, the function `removeUnderlyingMapping()` removes an underlying-to-wrapper relationship from mapping `wrapperOf`. However, it does not invoke contract `DStockWrapper`'s function `setUnderlyingEnabled()` to remove the underlying asset from the wrapper contract. If this function is not invoked simultaneously, the wrapper contract can still accept deposits of assets that should be removed.

```
188 /// @notice Remove a single underlying mapping (e.g., when disabled in the wrapper).
189 /// @dev This only updates the factory registry; wrapper is expected to be disabled via
        setUnderlyingEnabled(false).
190 function removeUnderlyingMapping(address underlying) external onlyRole(OPERATOR_ROLE) {
191   address w = wrapperOf[underlying];
192   if (w == address(0)) revert NotRegistered();
193   wrapperOf[underlying] = address(0);
194   emit UnderlyingUnmapped(underlying, w);
195 }
```

Listing 2.26: src/DStockFactoryRegistry.sol

```
413 function setUnderlyingEnabled(address token, bool enabled) external onlyRole(OPERATOR_ROLE) {
414   UnderlyingInfo storage info = underlyings[token];
415   if (info.decimals == 0) revert UnknownUnderlying();
416   info.enabled = enabled;
417   emit UnderlyingStatusChanged(token, enabled);
418 }
```

Listing 2.27: src/DStockWrapper.sol

Impact The contract `DStockWrapper` may allow users to interact with underlying assets that are no longer registered in the factory's mapping.

Suggestion Invoke the function `setUnderlyingEnabled()` to remove underlying assets from the wrapper.

2.2 Recommendation

2.2.1 Rename return parameter in the function `underlyingInfo()`

Status Fixed in `Version 2`

Introduced by `Version 1`

Description The function returns a parameter named `decimals`. This name shadows the existing function `decimals()`. Such naming conflicts can create confusion and potentially lead to unexpected behavior in certain contexts. It is recommended to rename this return parameter.

```
400 function underlyingInfo(address token)
401   external
402   view
403   returns (bool enabled, uint8 decimals, uint256 liquidToken)
404 {
405   UnderlyingInfo memory info = underlyings[token];
406   return (info.enabled, info.decimals, info.liquidToken);
407 }
```

Listing 2.28: src/DStockWrapper.sol

```
145 function decimals() public pure override returns (uint8) { return 18; }
```

Listing 2.29: src/DStockWrapper.sol

Suggestion Rename the return parameter in the function `underlyingInfo()`.

2.2.2 Remove redundant code in the contract `DStockWrapper`

Status Fixed in `Version 2`

Introduced by `Version 1`

Description There are several instances of unused code that should be removed for better maintainability.

1. Unused internal function `totalDebt_()`.

```
538 function totalDebt_() internal view returns (uint256) {
539   return _toAmount(_totalShares);
540 }
```

Listing 2.30: src/DStockWrapper.sol

2. Operations that will not be triggered. The contract does not invoke internal functions `_mint()` and `_burn()`, and the functions `transfer()` and `transferFrom()` validate addresses of sender and recipient are not zero-address. This makes the zero-address logic in `_update()` redundant.

```
239   if (from == address(0) || to == address(0)) {
240     super._update(from, to, value);
241     return;
242   }
```

Listing 2.31: src/DStockWrapper.sol

3. Unused parent contract. The contract inherits from the contract `UUPSUpgradeable`. However, the wrapper created using `createWrapper()` is a `BeaconProxy` and cannot be upgraded via `UUPS`.

```
109   wrapper = address(new BeaconProxy(address(beacon), initData));
110
111   // Bookkeeping for wrapper
112   isWrapper[wrapper] = true;
113   allWrappers.push(wrapper);
114   emit WrapperCreated(wrapper, _p.name, _p.symbol);
115
116   // Map each initial underlying -> wrapper
117   for (uint256 i = 0; i < _p.initialUnderlyings.length; i++) {
118     address u = _p.initialUnderlyings[i];
119     wrapperOf[u] = wrapper;
120     emit UnderlyingMapped(u, wrapper);
121   }
```

Listing 2.32: src/DStockFactoryRegistry.sol

```
27 UUPSUpgradeable
```

Listing 2.33: src/DStockWrapper.sol

```
115   __UUPSUpgradeable_init();
```

Listing 2.34: src/DStockWrapper.sol

```
426 function _authorizeUpgrade(address newImplementation) internal override onlyRole(UPGRADER_ROLE)
        {}
```

Listing 2.35: src/DStockWrapper.sol

```
34 bytes32 public constant UPGRADER_ROLE = keccak256("UPGRADER_ROLE");
```

Listing 2.36: src/DStockWrapper.sol

Suggestion Remove these redundant codes.

2.2.3 Validate new values in setter functions

Status Fixed in Version 2

Introduced by Version 1

Description Several setter functions emit change events without validating that new values differ from current ones, causing unnecessary event emissions and potential confusion for off-chain monitoring systems.

```
275 function setUnwrapFeeBps(uint16 bps) external onlyRole(OPERATOR_ROLE) {
276   uint16 old = unwrapFeeBps;
277   unwrapFeeBps = bps;
278   emit UnwrapFeeChanged(old, bps);
279 }
280
281 function setCap(uint256 newCap) external onlyRole(OPERATOR_ROLE) {
282   uint256 old = cap;
283   cap = newCap;
284   emit CapChanged(old, newCap);
285 }
286
287 function setTermsURI(string calldata uri) external onlyRole(OPERATOR_ROLE) {
288   string memory old = termsURI;
289   termsURI = uri;
290   emit TermsURIChanged(old, uri);
291 }
```

Listing 2.37: src/DStockWrapper.sol

```
257 function setCompliance(address c) external onlyRole(OPERATOR_ROLE) {
258   address old = address(compliance);
259   compliance = IDStockCompliance(c);
260   emit ComplianceChanged(old, c);
261 }
262
263 function setTreasury(address t) external onlyRole(OPERATOR_ROLE) {
264   address old = treasury;
265   treasury = t;
266   emit TreasuryChanged(old, t);
267 }
268
269 function setWrapFeeBps(uint16 bps) external onlyRole(OPERATOR_ROLE) {
270   uint16 old = wrapFeeBps;
271   wrapFeeBps = bps;
272   emit WrapFeeChanged(old, bps);
273 }
```

Listing 2.38: src/DStockWrapper.sol

```
141 function setGlobalCompliance(address c) external onlyRole(OPERATOR_ROLE) {
142   address old = globalCompliance;
143   globalCompliance = c; // can be zero to disable compliance checks by default
144   emit GlobalComplianceChanged(old, c);
```

```
145 }
```

Listing 2.39: src/DStockFactoryRegistry.sol

Suggestion Validate new values differ from the current state in setter functions

2.2.4 Lack of invoking function `_disableInitializers()`

Status Fixed in `Version 2`

Introduced by `Version 1`

Description In contract `DStockWrapper`, the function `_disableInitializers()` is not invoked in the constructor. Invoking this function prevents the contract itself from being initialized, thereby avoiding unexpected behaviors.

Suggestion Invoke the function `_disableInitializers()` in the constructor.

2.2.5 Implement logic for old wrapper after migration

Status Fixed in `Version 2`

Introduced by `Version 1`

Description In the contract `DStockFactoryRegistry`, the function `migrateUnderlyings()` enables the admin to migrate a batch of underlying assets from `oldWrapper` to `newWrapper`. However, there is no logic in processing `oldWrapper` and its underlying funds after migration.

```
188 /// @notice Remove a single underlying mapping (e.g., when disabled in the wrapper).
189 /// @dev This only updates the factory registry; wrapper is expected to be disabled via
        setUnderlyingEnabled(false).
190 function removeUnderlyingMapping(address underlying) external onlyRole(OPERATOR_ROLE) {
191   address w = wrapperOf[underlying];
192   if (w == address(0)) revert NotRegistered();
193   wrapperOf[underlying] = address(0);
194   emit UnderlyingUnmapped(underlying, w);
195 }
```

Listing 2.40: src/DStockFactoryRegistry.sol

```
413 function setUnderlyingEnabled(address token, bool enabled) external onlyRole(OPERATOR_ROLE) {
414   UnderlyingInfo storage info = underlyings[token];
415   if (info.decimals == 0) revert UnknownUnderlying();
416   info.enabled = enabled;
417   emit UnderlyingStatusChanged(token, enabled);
418 }
```

Listing 2.41: src/DStockWrapper.sol

Suggestion Implement logic to properly handle the `oldWrapper` after migration.

Feedback from the project The migration mechanism has been removed in order to simplify the implementation.

2.3 Note

2.3.1 Potential centralization risks

Introduced by Version 1

Description In this project, several privileged roles (e.g., `DEFAULT_ADMIN_ROLE`) can conduct sensitive operations, which introduces potential centralization risks. For example, operators can modify fees and control underlying token mappings without user consent. If the private keys of the privileged accounts are lost or maliciously exploited, it could pose a significant risk to the protocol.

2.3.2 Assumption of underlying token value equivalence

Introduced by Version 1

Description In the contract `DStockWrapper`, function `unwrap()` allows users to exchange their shares for any underlying token supported by the wrapper. The core mechanism of the wrapper relies on the implicit security assumption that all underlying tokens mapped to a single wrapper are exchangeable at an equivalent value (i.e., they are essentially fungible within the wrapper's context). This design allows the contract to manage liquidity and redemptions based on the collective pool of underlying tokens without considering individual market prices relative to each other. If this premise is violated, severe financial imbalance and asset drain may occur.

2.3.3 Potential risks regarding peg-out events

Introduced by Version 1

Description According to contract `DStockWrapper`, multiple underlying assets can point to a D-stock. D-stock shareholders can unwrap arbitrary underlying assets supported in the contract `DStockWrapper`. This introduces a potential risk of a bank run when one of the underlying assets is pegged out. The project and any involved parties should be aware of such risks.

2.3.4 Ensure consistent fee rates among assets in the same wrapper

Introduced by Version 2

Description Underlying assets in contract `DStockWrapper` may incur holding fees from both the underlying asset itself and the protocol. A concern arises when fee rates differ among underlying assets.

Specifically, the function `unwrap()` calculates the number of burned shares based on the proportion between all underlying assets and the asset amounts being redeemed. This proportional calculation does not account for individual fee rate differences. Consequently, when the wrapper contains underlying assets with different fee rates, users who deposited higher-fee assets gain an advantage. They effectively unwrap at an average rate lower than what they should pay. Conversely, users who deposited lower-fee assets are disadvantaged.

To mitigate this side effect, the project should ensure each wrapper contains only underlying assets with the same fee rate.

2.3.5 Time gaps in split and merge adjustments

Introduced by `Version 2`

Description Multiple underlying assets in a wrapper contract are pegged to the same real-world stock. When the stock undergoes a split or merge, the underlying asset should undergo corresponding balance adjustments. However, a time gap between different assets performing these adjustments can lead to significant differences in their values. Since the project's wrap and unwrap calculations rely on the equivalence of values among underlying assets, this scenario disrupts the functionality, allowing users to withdraw assets with disproportionate values relative to their deposits.

Feedback from the project The project decided to mitigate the issue by pausing the contracts in advance and invoking the function `applySplit()` to perform any necessary adjustments. The pause will remain in effect until all underlying assets have completed the split or merge process.


BOOST WEB3 THROUGH NEXT-GENERATION SECURITY & USABILITY INNOVATIONS