Report

v. 1.0

Customer

Ring Protocol

Smart Contract Audit

Hook

24th November 2025

Report prepared by

ABDK Consulting

Contents

1 Changelog

#	Date	Author	Description
0.1	24.11.25	D. Khovratovich	Initial Draft
0.2	24.11.25	D. Khovratovich	Minor revision
1.0	24.11.25	D. Khovratovich	Release

2 Summary

This document presents the results of a smart contract audit performed by ABDK Consulting for Ring Protocol's System of `FewETHHook`, `FewTokenHook`, and `FewUSDTHook` integrating with `Uniswap V4` periphery. The audit was conducted by Mikhail Vladimirov and Dmitry Khovratovich between 7th November and 19th November 2025. Our goal was to assess correctness and safety of 1:1 wrapping and unwrapping flows, delta settlement with `IPoolManager`, adherence to `BaseHook` permissions, reentrancy protection, token approvals, and validation/error semantics across the System.

Our conclusion is that the System demonstrates solid code quality and alignment with `Uniswap V4` hook patterns, using well-regarded dependencies (`OpenZeppelin IERC20`/`SafeERC20`, `solmate WETH`/`ReentrancyGuard`). The most impactful issues are Major logic checks around invariant enforcement and return-value handling.With the fixes confirmed as implemented, the maturity level is good and suitable for further testing and staged deployment.

Overview of important findings: Major issues include improper invariant checks comparing outputs to zero rather than to expected amounts in wrapping/unwrapping paths `CVF-1`, `CVF-2`, `CVF-3`, `CVF-5`, `CVF-6`; and ignoring the return value from unwrapping in `FewETHHook` `CVF-4`. Classes of findings were concentrated in Unclear behavior and Flaw. All Major items are marked Fixed. The codebase relies on `BaseHook`, `DeltaResolver`, `BeforeSwapDelta`, `SafeCast`, `Currency`/`CurrencyLibrary`, and `IFewWrappedToken`, matching the documented intent to provide 1:1 wrapping inside `beforeSwap`.

General recommendations: enforce exact 1:1 invariants in all wrappers and require checking return values from `wrap`/`unwrap`; preserve reentrancy guards and use `SafeERC20` where applicable; maintain immutability and clear visibility for state; add unit and integration tests for `beforeSwap` deltas, `_pay`/`_settle` interactions with `IPoolManager`, and initialization checks. Final verdict: the System is well-structured, consistent with `Uniswap V4` paradigms, and, after the addressed Major fixes, appears low risk for its stated 1:1 wrapping scope.

It is important to note that a security audit is not a guarantee of absolute security but rather a snapshot of the system's security posture at a specific point in time. While we strive to uncover all potential issues, the evolving nature of blockchain technology and DeFi means new vulnerabilities can emerge.

3 System overview

This section provides a high-level overview of the System that enables 1:1 wrapping and unwrapping of tokens within `Uniswap V4` liquidity pools using specialized hook contracts. The System's purpose is to automate conversion between underlying assets and their wrapped counterparts during pool interactions so that user operations proceed seamlessly while pool accounting with the `PoolManager` remains correct. The System is implemented through three concrete hook contracts `FewETHHook`, `FewTokenHook`, and `FewUSDTHook` that each target a specific asset pairing and follow the `Uniswap V4` hook model.

We were asked to review:

- Original Code
- Code with Fixes

Files:

src/hooks/

FewETHHook.sol	FewTokenHook.sol	FewUSDTHook.sol

Core components begin with the abstract base `BaseHook` in `src/utils/BaseHook.sol` which formalizes the permissions and lifecycle for `Uniswap V4` hooks. It wires an authorization boundary to the pool manager and exposes lifecycle entry points for pool initialization, liquidity changes, swaps, and donations. This base provides a declarative permissions structure that any concrete hook must declare so that the deployed hook address reflects the intended capabilities. The base validates the deployed address against those declared permissions.

The second foundational module is `DeltaResolver` in `src/base/DeltaResolver.sol` which encapsulates interactions with `IPoolManager` to synchronize balances, take owed credits, and settle debts for currencies tracked by the pool. It centralizes mapping of amounts for settlement or taking, ensures correct sign conventions for pool deltas, and emits descriptive errors for misuse such as attempting to settle a positive delta or take a negative one. This module abstracts the System's accounting with the pool manager and provides a predictable path for moving funds to and from the pool's accounting domain.

On top of these foundations, the System provides three production hooks in `src/hooks`. `FewETHHook` coordinates conversion between native ETH and a wrapped Few ETH token `fwWETH`. It uses `solmate WETH` to convert ETH and then interfaces with `IFewWrappedToken` to wrap and unwrap. The contract records the wrapper currency and the underlying currency, and it determines the wrapping direction based on token ordering. It restricts the target pool to the wrapper and its underlying with a zero fee, applies reentrancy protection, and pre-approves the wrapper for efficient operations.

FewTokenHook generalizes the same 1:1 wrapping pattern for a standard ERC-20 token and its Few wrapper. It integrates IERC20 for balance checks and approvals, stores immutable currency descriptors, and follows the same pool validation and lifecycle behavior as the ETH version. It employs the same accounting layer through DeltaResolver to synchronize with IPoolManager while keeping the conversion logic isolated to the wrapper interface IFewWrappedToken.

FewUSDTHook specializes the generic token approach for USDT. Because USDT's approval behavior differs from typical ERC-20 tokens, the contract uses SafeERC20 helpers to ensure approvals are safely established. The rest of the lifecycle mirrors the other hooks by narrowing pool composition to the wrapped token and its underlying with zero fee, recording immutable currency descriptors, and using reentrancy protection. The role of this module is to provide consistent 1:1 conversions while accommodating token specific approval semantics.

Relationships among modules are straightforward. Each production hook inherits BaseHook to declare hook permissions and lifecycle integration and inherits DeltaResolver to manage pool manager accounting. Hooks also mix in ReentrancyGuard for runtime safety. The hooks compose these features to offer a single responsibility layer that handles asset conversion at the correct moments in the pool lifecycle and then settles or takes balances according to pool deltas.

The System connects to external systems through well defined interfaces. It integrates with Uniswap V4 IPoolManager for balance synchronization and settlement. It relies on Currency typed wrappers for safe handling of ETH and tokens. It consumes ERC-20 tokens through IERC20 and uses SafeERC20 in the USDT specific case. For ETH conversions, it uses WETH from solmate. For asset wrapping, it speaks to IFewWrappedToken to produce or redeem wrapped units that track the underlying 1:1.

From a behavioral perspective the hooks validate that a pool configuration matches the intended pair and fee, then allow liquidity operations. During trading paths that imply wrapping or unwrapping, the hook transfers in the required asset, performs the conversion through the wrapper or WETH, and settles the resulting balances back to the pool manager. The design isolates conversion logic from settlement logic and establishes predictable control flow with clear errors if balances are insufficient or invariants are violated. This separation supports readability and maintainability.

The issues list in the CSV focuses on correctness and robustness of conversions and pool accounting. Major items highlight the need to compare output amounts to the expected inputs rather than to zero and to check return values from unwrapping when available. Minor items suggest visibility tuning, datatype preferences for immutables, and code organization refinements. The feedback aligns with the System's intent to enforce strict 1:1 behavior and predictable error handling. The emphasis on invariant checks, return value handling, and safe approvals informs testing priorities and strengthens assurance around conversions in live operation.

In summary, our team assessed a modular System that leverages the Uniswap V4 hook framework to provide deterministic 1:1 asset conversions at precise lifecycle points while deferring balance accounting to IPoolManager via DeltaResolver. The three hook implementations cover ETH, generic ERC-20, and USDT cases using consistent patterns and targeted dependencies. The architecture is cohesive and oriented to safe integration with external tokens and the pool manager, with explicit validations and runtime guards that align with the System's conversion purpose.

4 Methodology

The methodology is not a strict formal procedure, but rather a selection of methods and tactics combined differently and tuned for each particular project, depending on the project structure and technologies used, as well as on client expectations from the audit.

- **General Code Assessment**. The code is reviewed for clarity, consistency, style, and for whether it follows best code practices applicable to the particular programming language used. We check indentation, naming convention, commented code blocks, code duplication, confusing names, confusing, irrelevant, or missing comments etc. At this phase we also understand overall code structure.
- **Entity Usage Analysis**. Usages of various entities defined in the code are analysed. This includes both: internal usages from other parts of the code as well as potential external usages. We check that entities are defined in proper places as well as their visibility scopes and access levels are relevant. At this phase, we understand overall system architecture and how different parts of the code are related to each other.
- **Access Control Analysis**. For those entities, that could be accessed externally, access control measures are analysed. We check that access control is relevant and done properly. At this phase, we understand user roles and permissions, as well as what assets the system ought to protect.
- **Code Logic Analysis**. The code logic of particular functions is analysed for correctness and efficiency. We check whether the code actually does what it is supposed to do, whether the algorithms are optimal and correct, and whether proper data types are used. We also make sure that external libraries used in the code are up to date and relevant to the tasks they solve in the code. At this phase we also understand data structures used and the purposes they are used for.

We classify issues by the following severity levels:

- **Critical issue** directly affects the smart contract functionality and may cause a significant loss.
- **Major issue** is either a solid performance problem or a sign of misuse: a slight code modification or environment change may lead to loss of funds or data. Sometimes it is an abuse of unclear code behaviour which should be double checked.
- **Moderate issue** is not an immediate problem, but rather suboptimal performance in edge cases, an obviously bad code practice, or a situation where the code is correct only in certain business flows.
- **Recommendations** cover code style, best practices and general improvements.

5 Our findings

We found 6 major, and a few less important issues. All identified Major issues have been fixed.

Fixed 6 out of 6 issues

6 Major Issues

CVF-1 FIXED

- **Category** Unclear behavior
- **Source** FewUSDTHook.sol

Description The value should be compared with “underlyingAmount“ rather than with zero.

Client Comment *We agree this should be enhanced. For 1:1 wrapping, we should verify wrappedAmount == underlyingAmount. We will add:*

```
  if (wrappedAmount != underlyingAmount) revert WrapFailed();}
```

```
162 if (wrappedAmount == 0) revert WrapFailed();
```

CVF-2 FIXED

- **Category** Unclear behavior
- **Source** FewUSDTHook.sol

Description The value should be compared with “wrapperAmount“ rather than with zero.

Client Comment *We agree this should be enhanced. For 1:1 unwrapping, we should verify unwrappedAmount == wrapperAmount. We will add:*

```
if (unwrappedAmount != wrapperAmount) revert UnwrapFailed();}
```

```
177 if (unwrappedAmount == 0) revert UnwrapFailed();
```

CVF-3 FIXED

- **Category** Unclear behavior
- **Source** FewETHHook.sol

Description The value should be compared with “underlyingAmount“ rather than with zero.

Client Comment *We agree this should be enhanced. For 1:1 wrapping, we should verify wrappedAmount == underlyingAmount. We will add:*

```
  if (wrappedAmount != underlyingAmount) revert WrapFailed();}
```

```
165 if (wrappedAmount == 0) revert WrapFailed();
```

CVF-4 FIXED

- **Category** Flaw
- **Source** FewETHHook.sol

Description The returned value is ignored.

Recommendation Check the returned value to match “wrapperAmount”.

Client Comment *We agree this is a valid issue. The unwrap() function returns a value that should be checked. We will fix:*

```
fwWETH.unwrap(wrapperAmount);
uint256 unwrappedAmount = fwWETH.unwrap(wrapperAmount);
and verify: if (unwrappedAmount != wrapperAmount) revert
   ↪ UnwrapFailed();}
```

```
179 fwWETH.unwrap(wrapperAmount);
```

CVF-5 FIXED

- **Category** Unclear behavior
- **Source** FewTokenHook.sol

Description The value should be compared with “underlyingAmount“ rather than with zero.

Client Comment *We agree this should be enhanced. For 1:1 wrapping, we should verify wrappedAmount == underlyingAmount. We will add:*

```
  if (wrappedAmount != underlyingAmount) revert WrapFailed();}
```

```
159 if (wrappedAmount == 0) revert WrapFailed();
```

CVF-6 FIXED

- **Category** Unclear behavior
- **Source** FewTokenHook.sol

Description The value should be compared with “wrapperAmount“ rather than with zero.

Client Comment *We agree this should be enhanced. For 1:1 unwrapping, we should verify unwrappedAmount == wrapperAmount. We will add:*

```
  if (unwrappedAmount != wrapperAmount) revert UnwrapFailed();}
```

```
175 if (unwrappedAmount == 0) revert UnwrapFailed();
```

7 Moderate Issues

CVF-7 FIXED

- **Category** Bad datatype
- **Source** FewETHHook.sol

Description The “underlyingCurrency” variable should be turned into a constant.

Client Comment *The underlyingCurrency is set in the constructor based on runtime parameters (fewToken.token() for FewTokenHook, ADDRESS_ZERO for FewETHHook), so it cannot be a compile-time constant. The current immutable design is correct.*

```
41 /// @notice The underlying token currency (ETH)
   Currency public immutable underlyingCurrency;
```

```
64     underlyingCurrency = CurrencyLibrary.ADDRESS_ZERO;
```

8 Recommendations

CVF-8 INFO

- **Category** Procedural
- **Source** FewUSDTHook.sol

Description We didn't review these files.

Client Comment *Acknowledged. These are external dependencies that are well-audited libraries:*

- *BaseHook: https://github.com/Uniswap/v4-periphery/blob/main/src/utils/BaseHook.sol (Uniswap official implementation)*
- *DeltaResolver: Part of our codebase, but follows Uniswap V4 patterns*
- *ReentrancyGuard: From solmate (https://github.com/transmissions11/solmate), a widely-used and audited library*

```
6  import {ReentrancyGuard} from "solmate/src/utils/ReentrancyGuard.sol
       ↪ ";
```

```
15 import {BaseHook} from "../utils/BaseHook.sol";
   import {DeltaResolver} from "../base/DeltaResolver.sol";
   import {IFewWrappedToken} from "../interfaces/external/
       ↪ IFewWrappedToken.sol";
```

CVF-9 FIXED

- **Category** Suboptimal
- **Source** FewUSDTHook.sol

Description These errors could be made more useful by adding certain parameters into them.

Client Comment *We agree this would improve debugging. We will enhance errors with parameters:*

```
error WrapFailed(uint256 expected, uint256 actual);
error UnwrapFailed(uint256 expected, uint256 actual);}
```

```
31 error WrapFailed();
   error UnwrapFailed();
   error InvalidAddress();
```

CVF-10 INFO

- **Category** Suboptimal
- **Source** FewUSDTHook.sol

Description These checks are redundant, as it is anyway possible to pass a dead address.

Recommendation Remove these checks.

Client Comment *While dead addresses can still be passed, checking for address(0) is a best practice for input validation and provides clearer error messages. The gas cost is minimal.*

```
55 if (address(_manager) == address(0)) revert InvalidAddress();
   if (address(_fewUSDT) == address(0)) revert InvalidAddress();
```

CVF-11 INFO

- **Category** Suboptimal
- **Source** FewUSDTHook.sol

Description The “isExactInput” condition is checked twice.

Recommendation Refactor the code to check only once.

Client Comment *We acknowledge this could be optimized, but the current structure (branching by wrap vs unwrap) keeps the code clearer. The gas impact of reusing the boolean is negligible.*

```
124     isExactInput ? uint256(-params.amountSpecified) :
           ↪ _getWrapInputRequired(uint256(params.amountSpecified));
```

```
129     isExactInput ? -wrappedAmount.toInt256().toInt128() :
           ↪ inputAmount.toInt256().toInt128();
```

```
134 uint256 inputAmount = isExactInput
        ? uint256(-params.amountSpecified)
```

```
141     isExactInput ? -unwrappedAmount.toInt256().toInt128() :
           ↪ inputAmount.toInt256().toInt128();
```

CVF-12 INFO

- **Category** Suboptimal
- **Source** FewUSDTHook.sol

Description These two functions just return their argument values as they are.

Recommendation Consider removing these functions.

Client Comment *These functions are part of the BaseTokenWrapperHook interface and can be overridden for non-1:1 wrappers (like wstETH). Keeping them maintains interface consistency and allows future extensibility. The gas overhead is negligible.*

```
183 function _getWrapInputRequired(uint256 wrappedAmount) internal pure
       ↪ returns (uint256) {
        return wrappedAmount;
```

```
188 function _getUnwrapInputRequired(uint256 underlyingAmount) internal
       ↪ pure returns (uint256) {
        return underlyingAmount;
```

CVF-13 INFO

- **Category** Procedural
- **Source** FewETHHook.sol

Description We didn't review these files.

Client Comment *Acknowledged. These are external dependencies.*

```
4 import {WETH} from "solmate/src/tokens/WETH.sol";
```

```
15 import {BaseHook} from "../utils/BaseHook.sol";
import {DeltaResolver} from "../base/DeltaResolver.sol";
import {IFewWrappedToken} from "../interfaces/external/
   ↪ IFewWrappedToken.sol";
```

CVF-14 FIXED

- **Category** Suboptimal
- **Source** FewETHHook.sol

Description These error could be made ore useful by adding vertain parameters into them.
Client Comment *We will enhance errors with parameters:*

```
error WrapFailed(uint256 expected, uint256 actual);
error UnwrapFailed(uint256 expected, uint256 actual);}
```

```
30 error InvalidAddress();
   error WrapFailed();
   error UnwrapFailed();
```

CVF-15 FIXED

- **Category** Suboptimal
- **Source** FewETHHook.sol

Description These varaibles don't need to be public.
Client Comment *These can be changed to internal or removed if not needed externally. We will review and optimize visibility.*

```
35 WETH public immutable weth;
```

```
37 IFewWrappedToken public immutable fwWETH;
```

CVF-16 INFO

- **Category** Bad datatype
- **Source** FewETHHook.sol

Description The "wrapZeroForOne" variable should be turned into a constant.
Client Comment *Similar to Issue #8, this is calculated at runtime based on token address ordering, so it cannot be a compile-time constant. The immutable design is correct.*

```
43 /// @notice Indicates whether wrapping occurs when swapping from
      ↪ token0 to token1
   bool public immutable wrapZeroForOne;
```

```
64     underlyingCurrency = CurrencyLibrary.ADDRESS_ZERO;
       wrapZeroForOne = underlyingCurrency < wrapperCurrency;
```

CVF-17 FIXED

- **Category** Bad datatype
- **Source** FewETHHook.sol

Description The type for the "_weth" argument should be "WETH".

Client Comment *We agree this would improve type safety. However, we use address payable to match the WETH contract's interface. We can consider changing to WETH type if it doesn't break compatibility.*

```
53 constructor(IPoolManager _manager, address payable _weth,
       ↪ IFewWrappedToken _fwWETH)
```

CVF-18 INFO

- **Category** Suboptimal
- **Source** FewETHHook.sol

Description These checks are redundant, as it is anyway possible to pass a dead address.

Recommendation Remove these checks.

Client Comment *While dead addresses can still be passed, checking for address(0) is a best practice.*

```
57 if (address(_manager) == address(0)) revert InvalidAddress();
   if (_weth == address(0)) revert InvalidAddress();
   if (address(_fwWETH) == address(0)) revert InvalidAddress();
```

CVF-19 INFO

- **Category** Suboptimal
- **Source** FewETHHook.sol

Description The “isExactInput” condition is checked twice.

Recommendation Refactor the code to check only once.

Client Comment *We acknowledge this could be optimized, but the current structure (wrapping vs unwrapping branches) is clearer and more maintainable. The gas savings would be minimal.*

```
126     isExactInput ? uint256(-params.amountSpecified) :
           ↪ _getWrapInputRequired(uint256(params.amountSpecified));
```

```
131     isExactInput ? -wrappedAmount.toInt256().toInt128() :
           ↪ inputAmount.toInt256().toInt128();
```

```
136 uint256 inputAmount = isExactInput
        ? uint256(-params.amountSpecified)
```

```
143     isExactInput ? -unwrappedAmount.toInt256().toInt128() :
           ↪ inputAmount.toInt256().toInt128();
```

CVF-20 INFO

- **Category** Suboptimal
- **Source** FewETHHook.sol

Description The balance check is redundant, as insufficient balance will anyway cause the “deposit” call to fail.

Recommendation Remove the balance check.

Client Comment *These checks provide early failure with clear error messages before external calls, improving user experience. The gas cost is minimal.*

```
161 if (address(this).balance < underlyingAmount) revert
       ↪ InsufficientBalance();
```

```
163 weth.deposit{value: underlyingAmount}();
```

CVF-21 INFO

- **Category** Suboptimal
- **Source** FewETHHook.sol

Description The fwWETH balance check is redundant, as insufficient balance will anyway cause the “unwrap” call to fail.

Recommendation Remove the balance check.

Client Comment *These checks provide early failure with clear error messages before external calls, improving user experience. The gas cost is minimal.*

```
177 if (IERC20(address(fwWETH)).balanceOf(address(this)) < wrapperAmount
    ↪ ) revert InsufficientBalance();
```

```
179 fwWETH.unwrap(wrapperAmount);
```

CVF-22 FIXED

- **Category** Suboptimal
- **Source** FewETHHook.sol

Description The double type conversion is redundant, as “fwWETH” is of type “IFewWrappedToken” which already has the “balanceOf” function.

Client Comment *Since fwWETH is IFewWrappedToken which extends IERC20, we can use fwWETH.balanceOf() directly.*

```
177 if (IERC20(address(fwWETH)).balanceOf(address(this)) < wrapperAmount
    ↪ ) revert InsufficientBalance();
```

CVF-23 INFO

- **Category** Suboptimal
- **Source** FewETHHook.sol

Description These two functions just return their argument values as they are.

Recommendation Consider removing these functions.

Client Comment *These functions are part of the BaseTokenWrapperHook interface and can be overridden for non-1:1 wrappers (like wstETH). Keeping them maintains interface consistency and allows future extensibility. The gas overhead is negligible.*

```
186 function _getWrapInputRequired(uint256 wrappedAmount) internal pure
    ↪ returns (uint256) {
     return wrappedAmount;
```

```
191 function _getUnwrapInputRequired(uint256 underlyingAmount) internal
    ↪ pure returns (uint256) {
     return underlyingAmount;
```

CVF-24 FIXED

- **Category** Documentation
- **Source** FewETHHook.sol

Description It is a good practice to put a comment into an empty block to explain why the block is empty.

Client Comment *We will add: // Required to receive ETH from WETH.withdraw()*

```
196 receive() external payable {}
```

CVF-25 INFO

- **Category** Procedural
- **Source** FewTokenHook.sol

Description We didn't review these files.

Client Comment *Acknowledged. These imports come from Uniswap libraries.*

```
5  import {ReentrancyGuard} from "solmate/src/utils/ReentrancyGuard.sol
   ↪ ";
```

```
14 import {BaseHook} from "../utils/BaseHook.sol";
   import {DeltaResolver} from "../base/DeltaResolver.sol";
   import {IFewWrappedToken} from "../interfaces/external/
   ↪ IFewWrappedToken.sol";
```

CVF-26 FIXED

- **Category** Suboptimal
- **Source** FewTokenHook.sol

Description These errors could be made more useful by adding certain parameters into them.

Client Comment *We will enhance errors with parameters:*

```
error WrapFailed(uint256 expected, uint256 actual);
error UnwrapFailed(uint256 expected, uint256 actual);}
```

```
29 error InvalidAddress();
30 error WrapFailed();
   error UnwrapFailed();
```

CVF-27 INFO

- **Category** Suboptimal
- **Source** FewTokenHook.sol

Description These checks are redundant, as it is anyway possible to pass a dead address.

Recommendation Remove these checks.

Client Comment *While dead addresses can still be passed, checking for address(0) is a best practice*

```
53 if (address(_manager) == address(0)) revert InvalidAddress();
   if (address(_fewToken) == address(0)) revert InvalidAddress();
```

CVF-28 INFO

- **Category** Suboptimal
- **Source** FewTokenHook.sol

Description The “isExactInput” condition is checked twice.

Recommendation Refactor the code to check only once.

Client Comment *We acknowledge this could be optimized, but the current structure (wrapping vs unwrapping branches) is clearer and more maintainable. The gas savings would be minimal.*

```
120     isExactInput ? uint256(-params.amountSpecified) :
           ↪ _getWrapInputRequired(uint256(params.amountSpecified));

125     isExactInput ? -wrappedAmount.toInt256().toInt128() :
           ↪ inputAmount.toInt256().toInt128();

130 uint256 inputAmount = isExactInput
        ? uint256(-params.amountSpecified)

137     isExactInput ? -unwrappedAmount.toInt256().toInt128() :
           ↪ inputAmount.toInt256().toInt128();
```

CVF-29 INFO

- **Category** Suboptimal
- **Source** FewTokenHook.sol

Description These two functions just return their argument values as they are.

Recommendation Consider removing these functions.

Client Comment *These functions are part of the BaseTokenWrapperHook interface and can be overridden for non-1:1 wrappers (like wstETH). Keeping them maintains interface consistency and allows future extensibility. The gas overhead is negligible.*

```
181 function _getWrapInputRequired(uint256 wrappedAmount) internal pure
    ↪ returns (uint256) {
      return wrappedAmount;
```

```
186 function _getUnwrapInputRequired(uint256 underlyingAmount) internal
    ↪ pure returns (uint256) {
      return underlyingAmount;
```

ABDK
Consulting

About us

Established in 2016, is a leading service provider in the space of blockchain development and audit. It has contributed to numerous blockchain projects, and co-authored some widely known blockchain primitives like Poseidon hash function.

The ABDK Audit Team, led by Mikhail Vladimirov and Dmitry Khovratovich, has conducted over 300 audits of blockchain projects in Solidity, Rust, Circom, C++, JavaScript, and other languages.

Contact

Email

dmitry@abdkconsulting.com

Website

abdk.consulting

Twitter

twitter.com/ABDKconsulting

LinkedIn

linkedin.com/company/abdk-consulting