QuillAudits

AUDIT REPORT

September 2025

For

HUDDLE 01

Table of Content

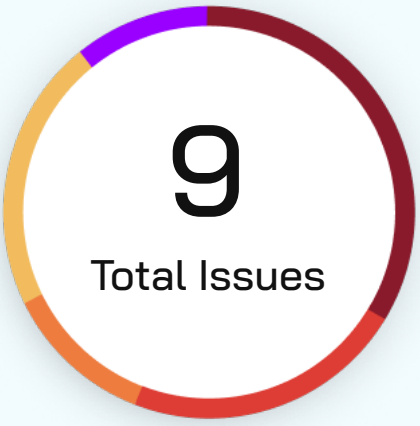
Executive Summary

Project Name	Huddle01
Protocol Type	Liquid Staking, Reward Distribution
Project URL	https://huddle01.com/
Overview	A liquid staking token wrapper for swETH with lockup periods with a reward distribution system for tHUDL tokens
Audit Scope	The scope of this Audit was to analyze the Huddle01 Smart Contracts for quality, security, and correctness.
Source Code link	https://github.com/Huddle01/hseth/tree/main/contracts/src
Branch	Main
Contracts in Scope	src/RewardDistributor.sol src/hsETH.sol src/interface/IswETH.sol
Commit Hash	2a9de8b2dda4c40274c7d36048faa4e8baee27b9
Language	Solidity
Blockchain	Ethereum
Method	Manual Analysis, Functional Testing, Automated Testing
Review 1	2nd September 2025 - 11th September 2025
Updated Code Received	11th September 2025 Commit : 89e36aa56a6b4413634658ca50825be0f12f7a3f
Review 2	15th September 2025 - 22ndSep 2025
Fixed In	https://github.com/Huddle01/hseth/tree/28e495cc4642db0bf539d88ca2c833bd74c8c0c8/contracts/src

Verify the Authenticity of Report on QuillAudits Leaderboard:

https://www.quillaudits.com/leaderboard

Number of Issues per Severity



Critical	3 (33.3%)
High	2 (22.2%)
Medium	1 (11.1%)
Low	2 (22.2%)
Informational	1 (11.1%)

Severity

Issues	Critical	High	Medium	Low	Informational
Open	0	0	0	0	0
Acknowledged	0	0	0	0	0
Partially Resolved	0	0	0	0	0
Resolved	**3**	**2**	**1**	**2**	**1**

Summary of Issues

Issue No.	Issue Title	Severity	Status
1	DoS Attack Through Deposit Limit Manipulation	Critical	Resolved
2	First Depositor Attack (Inflation Attack)	Critical	Resolved
3	Impossible Withdrawals	Critical	Resolved
4	Lockup Period Bypass via Transfer	High	Resolved
5	Owner Can Drain All Funds	High	Resolved
6	Exchange Rate Manipulation Risk	Medium	Resolved
7	rewardSigner will always return wrong data	Low	Resolved
8	UserDeposit[] array grows indefinitely	Low	Resolved
9	Centralization Risk	Informational	Resolved

Checked Vulnerabilities

- Access Management
- Arbitrary write to storage
- Centralization of control
- Ether theft
- Improper or missing events
- Logical issues and flaws
- Arithmetic Computations Correctness
- Race conditions/front running
- SWC Registry
- Re-entrancy
- Timestamp Dependence
- Gas Limit and Loops
- Exception Disorder
- Gasless Send
- Use of tx.origin
- Malicious libraries
- Compiler version not fixed
- Address hardcoded
- Divide before multiply
- Integer overflow/underflow
- ERC's conformance
- Dangerous strict equalities
- Tautology or contradiction
- Return values of low-level calls

- Missing Zero Address Validation
- Private modifier
- Revert/require functions
- Multiple Sends
- Using suicide
- Using delegatecall
- Upgradeable safety
- Using throw
- Using inline assembly
- Style guide violation
- Unsafe type inference
- Implicit visibility level

Techniques and Methods

Throughout the audit of smart contracts, care was taken to ensure:

- The overall quality of code
- Use of best practices
- Code documentation and comments, match logic and expected behavior
- Token distribution and calculations are as per the intended behavior mentioned in the whitepaper
- Implementation of ERC standards
- Efficient use of gas
- Code is safe from re-entrancy and other vulnerabilities

The following techniques, methods, and tools were used to review all the smart contracts:

Structural Analysis

In this step, we have analyzed the design patterns and structure of smart contracts. A thorough check was done to ensure the smart contract is structured in a way that will not result in future problems.

Static Analysis

A static Analysis of Smart Contracts was done to identify contract vulnerabilities. In this step, a series of automated tools are used to test the security of smart contracts.

Code Review / Manual Analysis

Manual Analysis or review of code was done to identify new vulnerabilities or verify the vulnerabilities found during the static analysis. Contracts were completely manually analyzed, their logic was checked and compared with the one described in the whitepaper. Besides, the results of the automated analysis were manually verified.

Gas Consumption

In this step, we have checked the behavior of smart contracts in production. Checks were done to know how much gas gets consumed and the possibilities of optimization of code to reduce gas consumption.

Tools and Platforms Used for Audit

Remix IDE, Foundry, Solhint, Mythril, Slither, Solidity Static Analysis.

Types of Severity

Every issue in this report has been assigned to a severity level. There are five levels of severity, and each of them has been explained below.

Critical: Immediate and Catastrophic Impact

Critical issues are the ones that an attacker could exploit with relative ease, potentially leading to an immediate and complete loss of user funds, a total takeover of the protocol's functionality, or other catastrophic failures. Critical vulnerabilities are non-negotiable; they absolutely must be fixed.

High (H): Significant Risk of Major Loss or Compromise

High-severity issues represent serious weaknesses that could result in significant financial losses for users, major malfunctions within the protocol, or substantial compromise of its intended operations. While exploiting these vulnerabilities might require specific conditions to be met or a moderate level of technical skill, the potential damage is considerable. These findings are critical and should be addressed and resolved thoroughly before the contract is put into the Mainnet.

Medium (M): Potential for Moderate Harm Under Specific Circumstances

Medium-severity bugs are loopholes in the protocol that could lead to moderate financial losses or partial disruptions of the protocol's intended behavior. However, exploiting these vulnerabilities typically requires more specific and less common conditions to occur, and the overall impact is generally lower compared to high or critical issues. While not as immediately threatening, it's still highly recommended to address these findings to enhance the contract's robustness and prevent potential problems down the line.

Low (L): Minor Imperfections with Limited Repercussions

Low-severity issues are essentially minor imperfections in the smart contract that have a limited impact on user funds or the core functionality of the protocol. Exploiting these would usually require very specific and unlikely scenarios and would yield minimal gain for an attacker. While these findings don't pose an immediate threat, addressing them when feasible can contribute to a more polished and well-maintained codebase.

Informational (I): Opportunities for Improvement, Not Immediate Risks

Informational findings aren't security vulnerabilities in the traditional sense. Instead, they highlight areas related to the clarity and efficiency of the code, gas optimization, the quality of documentation, or adherence to best development practices. These findings don't represent any immediate risk to the security or functionality of the contract but offer valuable insights for improving its overall quality and maintainability. Addressing these is optional but often beneficial for long-term health and clarity.

Types of Issues

Open

Security vulnerabilities identified that must be resolved and are currently unresolved.

Resolved

These are the issues identified in the initial audit and have been successfully fixed.

Acknowledged

Vulnerabilities which have been acknowledged but are yet to be resolved.

Partially Resolved

Considerable efforts have been invested to reduce the risk/impact of the security issue, but are not completely resolved.

Severity Matrix

Impact

Likelihood	High	Medium	Low
High	Critical	High	Medium
Medium	High	Medium	Low
Low	Medium	Low	Low

Impact

- **High** - leads to a significant material loss of assets in the protocol or significantly harms a group of users.
- **Medium** - only a small amount of funds can be lost (such as leakage of value) or a core functionality of the protocol is affected.
- **Low** - can lead to any kind of unexpected behavior with some of the protocol's functionalities that's not so critical.

Likelihood

- High - attack path is possible with reasonable assumptions that mimic on-chain conditions, and the cost of the attack is relatively low compared to the amount of funds that can be stolen or lost.
- Medium - only a conditionally incentivized attack vector, but still relatively likely.
- Low - has too many or too unlikely assumptions or requires a significant stake by the attacker with little or no incentive.

Critical Severity Issues

DoS Attack Through Deposit Limit Manipulation

Resolved

Path

hsETH.sol

Function Name

`deposit()`

Description

Fix introduces MAX_DEPOSITS_PER_USER, however an attacker can prevent any user from depositing by transferring small amounts to fill their deposit array.

```
function _update(
    address from,
    address to,
    uint256 amount
) internal virtual override {
    if (from != address(0) && to != address(0) && amount > 0) {
        _validateAndUpdateDepositsShares(from, amount);
        userDeposits[to].push(
            UserDeposit({shares: amount, depositTime: block.timestamp})
        ); // @note here
    }

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

deposit():

```
function deposit(
    uint256 amount
) external nonReentrant returns (uint256 sharesMinted) {
    if (amount == 0) revert InsufficientAmount();
    address user = _msgSender();

    uint256 activeDeposits = _getActiveDepositCount(user);
    if (activeDeposits >= MAX_DEPOSITS_PER_USER) { // @audit
        revert MaxDepositsReached();
    }
```

Attack Scenario

- Attacker transfers small amounts to victim

```
for(uint I = 0; I < 50; I++) {
    attacker.transfer(victim, 1); // Each creates a UserDeposit entry for
victim
}
```

- Now victim cannot deposit anymore

```
victim.deposit(1 ether); // Reverts: MaxDepositsReached()
```

Note these also affects areas where iterations are performed on user deposit array

Impact

Permanent denial of service for targeted users, protocol becomes unusable.

Likelihood

High

Recommendation

Consider addressing this on the transfer function or a different implementation.

First Depositor Attack (Inflation Attack)

Resolved

Path

hsETH.sol

Function Name

`deposit()`

Description

The first depositor can manipulate the share price to cause subsequent deposits to fail or receive zero shares.

Attack Scenario

Step 1: Attacker deposits 1 wei, gets 1 share

```
attacker.deposit(1); // sharesMinted = 1, totalSupply = 1, totalSwETH = 1
```

Step 2: Attacker directly transfers large amount to contract (bypassing deposit)

```
swETH.transfer(hsETHContract, 1000 ether); // totalSwETH = 1000 ether + 1 wei
```

Step 3: Next user tries to deposit

```
victim.deposit(999 ether);
// sharesMinted = (999 ether * 1) / (1000 ether + 1) = 0 (rounded down)
// Transaction reverts: "Deposit too small"
```

Step 4: Attacker withdraws their 1 share

```
attacker.withdraw(1);
```

`// Gets: (1 * totalSwETH) / 1 = ~1999 ether` (Gets back all his funds or steals victim's deposit if available)

Impact

Denial of service for legitimate users and potential Complete theft of user deposits if not addressed.

Likelihood

High

Recommendation

Protocols address these by mint a minimum amount to zero address, add a minimum deposit requirement, and virtual shares, etc. Consider the best approach to addressing this.

Impossible Withdrawals

Resolved

Path

hsETH.sol

Function Name

`withdraw()`

Description

The withdrawal system has a fundamental design flaw that makes it impossible for users to claim their funds in normal operation. The liquidity check incorrectly treats queued withdrawals as "unavailable" funds, even though those funds belong to the users trying to withdraw.

Root Cause - Flawed Liquidity Logic:

```
function claim(uint256 requestId) external nonReentrant {
    require(requestId < withdrawQueue.length, "invalid id");
    WithdrawRequest storage r = withdrawQueue[requestId];
    if (r.processed) revert NothingToProcess();
    if (r.user != msg.sender) revert NotRequestOwner();
    if (block.timestamp < r.unlockAt) revert RequestNotReady(r.unlockAt);

    uint256 available = _availableSwETH();
    if (available < r.swETHAmount) revert InsufficientLiquidity(); // @audit Critical logic failure

    r.processed = true;
```

_availableSwETH:

```
function _availableSwETH() internal view returns (uint256) {
    // Use actual ERC20 balance for liquidity checks to prevent claims without funds
    uint256 actualBalance = totalSwETH();
    if (actualBalance <= totalQueuedSwETH) return 0;
    return actualBalance - totalQueuedSwETH; // @audit WRONG: Treats user's own funds as "unavailable" when called in claim()
}
```

The Fundamental Problem:

The system incorrectly assumes that totalQueuedSwETH represents funds that are "unavailable" for withdrawals. However, totalQueuedSwETH represents funds that users are entitled to withdraw - they should be available for claims, not treated as unavailable.

Impact

Complete system failure

Total loss of user trust and protocol functionality

Likelihood

High

Recommendation

Consider implementing a fix that allows users to be able to withdraw their funds

High Severity Issues

Lockup Period Bypass via Transfer

Resolved

Path
hsETH.sol

Function
`withdraw()`

Description
The way the withdraw() function is currently implemented, anyone can easily bypass their lockup period by transferring the amount of hsETH they want to withdraw to their other addresses that never interacted with the deposit function of the contract and simply call withdraw using these other addresses.

Impact
The lockup period validation can be completely bypassed by transferring hsETH tokens between addresses

Likelihood
High

Recommendation
You might need to override the _transfer() function and perform some custom implementation on it to prevent this attack or implement a global lockup system

Owner Can Drain All Funds

Resolved

Path

hsETH.sol

Function

`emergencyWithdraw()`

Description

The emergencyWithdraw() function:

```solidity
function emergencyWithdraw(address token, uint256 amount) external onlyOwner {
    SafeERC20.safeTransfer(IERC20(token), owner(), amount);
}
```

As you can see, the Owner can steal all user deposits at any time by simply calling emergencyWithdraw(address(swETH), totalSwETH()). No limitations whatsoever.

Impact

Owner can drain all funds

Likelihood

High

Recommendation

Either remove this function or add proper emergency conditions and governance delays.

Medium Severity Issues

Exchange Rate Manipulation Risk

Resolved

Path

hsETH.sol

Path

`deposit()`

Description

The 1:1 exchange rate between swETH and hsETH doesn't account for potential slippage or fees:

```solidity
hsETHAmount = amount; // Always 1:1, ignores actual received amount
```

Impact

If swETH transfer involves fees or rebasing, users could receive incorrect hsETH amounts.

Likelihood

Medium

Recommendation

Check actual balance changes:

```solidity
uint256 balanceBefore = swETH.balanceOf(address(this));
SafeERC20.safeTransferFrom(IERC20(address(swETH)), user, address(this), amount);
uint256 balanceAfter = swETH.balanceOf(address(this));
hsETHAmount = balanceAfter - balanceBefore;
```

Low Severity Issues

rewardSigner will always return wrong data

Resolved

Path

RewardDistributor.sol

Description

The RewardDistributor contract declares a rewardSigner variable but is never initialized or modified anywhere in the contract. However, it still gets read from and returned in the getConfiguration() function. This leads to potential internal malfunctions and unexpected behavior from intended functionality.

Impact

Leads to potential internal malfunctions and unexpected behavior from intended functionality.

Likelihood

Low

Recommendation

Either implement signature-based claiming or remove the variable.

UserDeposit[] array grows indefinitely

Resolved

Description

The UserDeposit[] array grows indefinitely as users deposit. This is also due to the fact that there is no cleanup of the zero-amount entries from the array after being used and as the user keeps using the protocol, the _validateAndUpdateDeposits() has to iterate through the entire history, even both already used deposits.

Impact

Could lead to DOS via gas limit

Likelihood

Low

Recommendation

Implement deposit cleanup or use a different data structure.

Informational Issues

Centralization Risk

Resolved

Description

Contracts have owners with privileged rights to perform admin tasks on multiple sensitive functions and need to be trusted not to perform malicious updates or drain funds. The functions are: updateToken(), mintReward(), batchMintRewards(), updateSwETHContract(), updateLockupPeriod(), emergencyWithdraw().

Functional Tests

Some of the tests performed are mentioned below:

- Revert on zero amount deposits
- Withdrawal validation checks deposit timestamps
- Owner can update lockup period
- Revert when user has insufficient hsETH balance
- Confirm deposit tracking in userDeposits array
- Successfully withdraw unlocked hsETH tokens
- Multiple deposits track individual lockup periods
- Reverts on zero amount minting

Automated Tests

No major issues were found. Some false positive errors were reported by the tools. All the other issues have been categorized above according to their level of severity.

Threat Model

Contract	Function	Threats
hsETH	deposit()	External token calls could enable reentrancy
	withdraw()	Users can transfer tokens to bypass withdrawal restrictions
	updateSwETHContract()	**Owner could point to malicious swETH contract**
	withdraw()	Potential to withdraw more than deposited through manipulation
	_validateAndUpdateDeposits()	Large deposit arrays could cause out-of- gas failures
	emergencyWithdraw()	**Owner can steal all user deposits without restrictions**
RewardDistributor	mintReward()	Potential silent signature failure
	batchMintRewards()	Large batches could exceed block gas limit
	batchMintRewards()	Batch failures affect entire operation
	mintReward()	Low- level calls don’t verify token supports minting
	updateToken()	**Owner could point to malicious token contract**
	initialize()	Race condition during deployment

Closing Summary

In this report, we have considered the security of Huddle01. We performed our audit according to the procedure described above.

Issues of critical , high , medium , low and informational were found, at the end the Huddle01 team resolved all the issues.

Disclaimer

At QuillAudits, we have spent years helping projects strengthen their smart contract security. However, security is not a one-time event—threats evolve, and so do attack vectors. Our audit provides a security assessment based on the best industry practices at the time of review, identifying known vulnerabilities in the received smart contract source code.

This report does not serve as a security guarantee, investment advice, or an endorsement of any platform. It reflects our findings based on the provided code at the time of analysis and may no longer be relevant after any modifications. The presence of an audit does not imply that the contract is free of vulnerabilities or fully secure.

While we have conducted a thorough review, security is an ongoing process. We strongly recommend multiple independent audits, continuous monitoring, and a public bug bounty program to enhance resilience against emerging threats.

Stay proactive. Stay secure.

About QuillAudits

QuillAudits is a leading name in Web3 security, offering top-notch solutions to safeguard projects across DeFi, GameFi, NFT gaming, and all blockchain layers. With seven years of expertise, we've secured over 1400 projects globally, averting over $3 billion in losses. Our specialists rigorously audit smart contracts and ensure DApp safety on major platforms like Ethereum, BSC, Arbitrum, Algorand, Tron, Polygon, Polkadot, Fantom, NEAR, Solana, and others, guaranteeing your project's security with cutting-edge practices.

7+ Years of Expertise	**1M+** Lines of Code Audited
50+ Chains Supported	**1400+** Projects Secured

Follow Our Journey

AUDIT REPORT

September 2025

For

HUDDLE 01

QuillAudits

Canada, India, Singapore, UAE, UK

www.quillaudits.com audits@quillaudits.com