QuillAudits

AUDIT REPORT

October 2025

For

SurfLayer

Table of Content

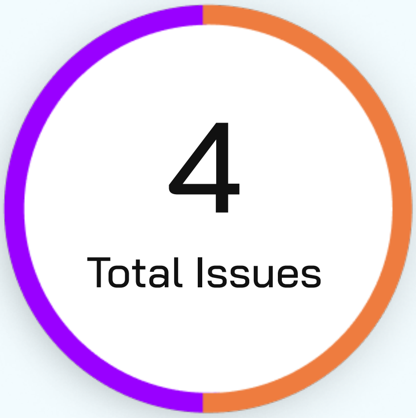
Executive Summary

Project Name	Surflayer
Protocol Type	Router
Project URL	https://surflayer.xyz
Overview	The SurfLayer consists of a single contract with distinct purpose. SurfRouter facilitates batch and single transfers of both native ETH and ERC20 tokens, emitting events to record all payments.
Audit Scope	The scope of this Audit was to analyze the SurfLayer Smart Contracts for quality, security, and correctness.
Source Code Link	Zip file was provided to QuillAudits Team https://drive.google.com/file/d/1nhUUCXwPV6pqDRw_mJs_NVoMX0p9WnE7/view?usp=drive_link
Contracts in Scope	SurfRouter.sol
Language	Solidity
Blockchain	EVM
Method	Manual Analysis, Functional Testing, Automated Testing
Review 1	7th October 2025 - 8th October 2025
Updated Code Received	9th September 2025
Review 2	9th September 2025
Fixed In	https://drive.google.com/file/d/1lD-83izHM42NTwNwcqKQ93hAniDh64A2/view?usp=drive_link

Verify the Authenticity of Report on QuillAudits Leaderboard:

https://www.quillaudits.com/leaderboard

Number of Issues per Severity



Critical	0 (0.0%)
High	0 (0.0%)
Medium	2 (50.0%)
Low	0 (0.0%)
Informational	2 (50.0%)

Severity

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

Summary of Issues

Issue No.	Issue Title	Severity	Status
1	Strict inequality can result in DOS under certain conditions	Medium	Acknowledged
2	Possibility of gas griefing by malicious address	Medium	Acknowledged
3	Lack of zero address check	Informational	Resolved
4	Consider using reentrancy guard from OZ	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

- [x] Missing Zero Address Validation
- [x] Private modifier
- [x] Revert/require functions
- [x] Multiple Sends
- [x] Using suicide
- [x] Using delegatecall
- [x] Upgradeable safety
- [x] Using throw
- [x] Using inline assembly
- [x] Style guide violation
- [x] Unsafe type inference
- [x] 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, F oundry, 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.

Medium Severity Issues

Strict inequality can result in DOS under certain conditions

Acknowledged

Path

SurfRouter.sol

Path

`sends()`

Description

The functions sends(), relies on strict inequality.
Consider sends() for an instance. It reverts is there is still total amount left in the function after the execution is done.

This can create a condition where even 1 extra wei can result in the function reverting. This might create conditions where even small roundings, miscalculation in frontend will result in the function being inability to be used.
Similar pattern can be seen in the other functions.

Impact

Temporary DOS of the function

Likelihood

Medium

Recommendation

Accept msg.value > sum and refund remaining value to the user.

Surflayer Teams' Comment

We want to ensure that the funds are transferred with 100% accuracy, guaranteeing that the input and output amounts are exactly the same.

Possibility of gas griefing by malicious address or DOS

Acknowledged

Path
SurfRouter.sol

Path
`sendNativeTransfer()`

Description
The function uses a low level call with 100k gas to call the target address. This design decision can manifest into gas griefing where a malicious address can consume all the input gas from the user.

On the other side, it can also result in DOS in a case where legitimate execution on the target address requires more than 100k gas.

Impact
Denial of service

Likelihood
Medium

Recommendation
Don’t push ETH to untrusted addresses. Instead, credit an internal balance and let recipients withdraw themselves via withdraw(). That prevents one bad recipient blocking others otherwise do not revert on failure of a single address.

Surflayer Teams’ Comment
We need to make sure all the funds will be transferred successfully. If have errors, we need to revert them all. To keep the funding safe.

Informational Issues

Lack of zero address check

Resolved

Path

SurfRouter.sol

Description

The contract does not validate against the use of the zero address (address(0)) in critical contexts:

Constructor / Ownership Initialization – If ownership is assigned to the zero address during deployment (e.g., via a typo, misconfigured script, or malicious initialization), the contract becomes permanently ownerless. This disables all privileged functions and may lock governance or upgrade mechanisms.

ETH and ERC20 Transfers – In functions like sends(), send(), sendTokens(), and sendToken(), recipients are not checked against the zero address. Sending ETH to address(0) results in permanent loss of funds. Sending ERC20 tokens to address(0) is equivalent to burning tokens, which may be unintentional and reduce user balances or token supply.

Recommendation

Consider checking for address(0) wherever necessary

Consider using reentrancy guard from OZ

Resolved

Path

SurfRouter.sol

Path

`nonReentrant()`

Description

The guard uses bool private locked; and toggles true/false.If the contract is intended to be used with proxies/upgrades, a simple bool might collide with storage layout in an upgrade; standard ReentrancyGuard from OZ uses a uint256 and constants that are less error prone.

Recommendation

Consider using OZ's reentrancy guard

Functional Tests

Some of the tests performed are mentioned below:

- ✔ Should send ETH to a single recipient via send
- ✔ Should send ETH to multiple recipients via sends with correct distribution
- ✔ Should revert sends if recipients and values array lengths mismatch
- ✔ Should emit SurfPayment event on successful ERC20 transfer

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

SurfRouter.sol	sends(address[] calldata tos, uint256[] memory values)	**Inputs**
		• tos
		  • Control: Fully controlled by the user.
		  • Constraints: Must match length of values.
		  • Impact: Determines the list of ETH recipients.
		• values
		  • Control: Fully controlled by the user.
		  • Constraints: Must match length of tos. Sum must equal msg.value.
		  • Impact: Determines the amount of ETH each recipient will receive.
		• msg.value
		  • Control: Fully controlled by the user.
		  • Constraints: Must equal sum of all values.
		  • Impact: The ETH actually provided for distribution.
		
		Branches and code coverage
		Intended branches
		• Should transfer the exact ETH amounts to each tos[I].
		• Should emit SurfPayment events for each successful transfer.

Contract	Function	Threats
		Test coverage • Should handle multiple recipients correctly. • Should revert if array lengths mismatch. • Should revert if total distributed > msg.value. • Should revert if leftover ETH remains (ExtraCostsExist). **Negative behavior** • Should not allow overfunding or underfunding. • Should not allow empty • arrays. • Should not allow sending to zero address (currently not enforced). • Should not allow bypassing ETH transfer failure.

	sendTokens(IERC20 token, address[] calldata tos, uint256[] memory values)	Allows a user to send ERC20 tokens to multiple recipients.
		Inputs
		• token
		  • Control: Fully controlled by the user.
		  • Constraints: Must be a valid ERC20 contract.
		  • Impact: Determines the token being distributed.

Contract	Function	Threats

		• tos
		  • Control: Fully controlled by the user.
		  • Constraints: Must match values.length.
		  • Impact: Determines recipients of tokens.
		• values
		  • Control: Fully controlled by the user.
		  • Constraints: Must not exceed user allowance or balance.
		  • Impact: Amount of tokens transferred per recipient.
		Branches and code coverage
		Intended branches
		  • Should transfer tokens to each tos(I) using safeTransferFrom.
		  • Should emit SurfPayment for each transfer.
		Test Coverage
		  • Should revert if msg.value > 0. Should revert if tos.length != values.length.
		  • Should transfer tokens correctly across multiple recipients.

Contract	Function	Threats
		Negative behavior • Should not allow sending to zero address. • Should not allow transfer amounts greater than balance/allowance. • Should not assume compatibility with fee-on-transfer tokens.

	send(address to)	**Inputs**
		• to
		  • Control: Fully controlled by the user.
		  • Constraints: None enforced.
		  • Impact: Receives ETH amount equal to msg.value.
		• msg.value
		  • Control: Fully controlled by the user.
		  • Constraints: Must be > 0.
		  • Impact: Amount of ETH transferred.
		Branches and code coverage
		Intended branches
		• Should transfer ETH to recipient.
		• Should emit SurfPayment.
		Negative Behaviour
		• Should not allow sending ETH to zero address.
		• Should not allow ETH transfer failure to pass silently.

Contract	Function	Threats

	sendToken(IERC20 token, address to, uint256 value)	Transfers ERC20 tokens to a single recipient.
		
		Inputs
		• token
		  • Control: Fully controlled by the user.
		  • Constraints: Must be valid ERC20.
		• to
		  • Control: Fully controlled by the user.
		  • Constraints: None enforced.
		• value
		  • Control: Fully controlled by the user.
		  • Constraints: Must not exceed sender balance/ allowance.
		
		Branches and code coverage
		Intended branches
		• Should transfer tokens to recipient.
		• Should emit SurfPayment.
		Negative Behaviour
		• Should not allow msg.value > 0.
		• Should not allow sending tokens to zero address.
		• Should not allow exceeding allowance/balance.

Closing Summary

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

The Surflayer team resolved two issues and acknowledged the other two 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.

QuillAudits

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

Follow Our Journey

AUDIT REPORT

October 2025

For

SurfLayer

QuillAudits

Canada, India, Singapore, UAE, UK

www.quillaudits.com audits@quillaudits.com