

# Security Document: Sharpe Base

*Written by:*

Thorat, Anirudha and Sagar, Sudeep

August 22, 2023

# 1 Executive Summary

This security report provides a detailed analysis and review of the Sharpe Base, a DeFi Platform that allows users to create, customize, and manage yield positions. The audit covers critical aspects, including the structure and design of Base Account, the architecture of Sharpe Base, and various security measures employed. Key findings include the identification and resolution of specific issues related to security and design, and the application of best practices such as static analysis with Slither, code coverage, and peer code reviews. The report highlights both the strengths and areas of improvement, offering insights and recommendations to enhance the overall security and quality of the code.

## 2 Methodology

The security audit of Sharpe Base was conducted using a systematic and rigorous approach, encompassing the following key areas:

- **Static Analysis:** Utilizing Slither, a static analysis framework designed specifically for Solidity smart contracts, to detect vulnerabilities, design flaws, and other issues.
- **Code Coverage:** Employing solidity-coverage to assess the effectiveness of test suites, identify untested code areas, and ensure high code quality.
- **Issue Analysis:** Investigating and categorizing issues identified in Base Account and Base Platform, including the analysis of false positives and triaging.
- **Peer Code Reviews:** Leveraging GitHub pull requests for peer code reviews to ensure adherence to coding standards, identify defects, and promote knowledge sharing.

The audit methodology ensured a comprehensive examination of the codebase, focusing on security by architecture and best practices, and aligning with industry standards and guidelines. The process facilitated a thorough understanding of potential risks, leading to actionable recommendations and continuous improvement in code quality and security.

## 3 Base Account

Any interaction with Sharpe Base requires a base-account. Any transaction requires the funds to be moved to a base-account either to bundle multiple transactions to one or to create/manage leveraged positions.

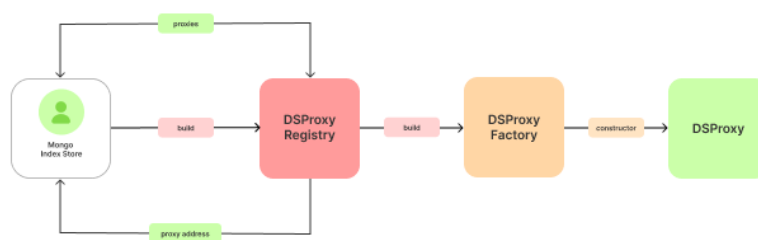
### 3.1 Usage

The user address is the sole owner of the base-account. In the later stages of the project, the user may authenticate certain automation contracts that shall allow the automation to manage his position on behalf of the user. This way, the user has complete control of his position at the individual level rather than a vault level.

**Note:** As per design, each user address is allowed to own only one instance of an account. Only after the account ownership is changed, the user address shall be able to create another account. Only the user shall be able to access any assets or manage the position in the account.

## 4 Components

### 4.1 Figure 1: User Builds a Base Account Instance for Self



## 4.2 Figure 2: User Sends Bundled Transaction Data to Sharpe Base



## 5 Contracts

### 5.1 DSProxy.sol

It is a simple contract that shall allow users to bundle tasks together like flashloan, swap, lend, and borrow using Sharpe Base.

### 5.2 DSProxyFactory.sol

It is the contract responsible for creating an instance of DSProxy for a user address.

### 5.3 DSProxyRegistry.sol

This contract uses DFProxyFactory to create an account and stores the handle to the account instance for each user only to retrieve it on demand.

## 6 Base

Sharpe Base is a DeFi Platform that allows users to create, customize, and manage yield positions.

### 6.1 Security By Architecture

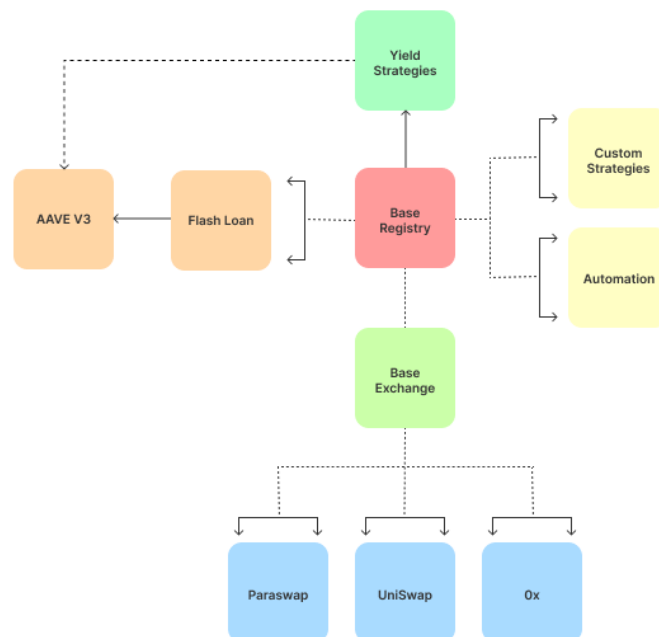


Figure 1: Base Registry and other components

#### 6.1.1 Overview

There are a couple of aspects of how security is being addressed by architecture and design:

**Absence of an ERC4626 Vault** It is a common design in the Ethereum world to create an ERC4626 Vault to earn yields. More about ERC4626 Vault can be seen [here](#).

**Not a Honey Pot** While an ERC4626 is an optimized and standard way to earn yields, it is also a honey pot for the attackers.

The architecture of Sharpe Base as a platform allows users to enter leverage positions and earn yields on popular protocols like AAVE via individual positions instead of a vault. The positions are held and managed by individual wallet addresses and their proxy accounts, i.e., smart contracts (Base Account: a proxy account that lets users program and automate or manually manage positions. More on this in the later section). Thus, an individual is able to create and manage leveraged wstETH - ETH positions without the need for an ERC4626 Vault.

**Sole Ownership and Position Management** An individual can now create and manage leverage positions on protocols like AAVE without the dependency of a Vault. The Base Account is a proxy smart contract solely managed by the user. It is designed to bundle multiple transactions like taking a flashloan, swap, buy/sell assets, lend, borrow, withdraw into a single transaction, which allows a novice DeFi user to earn higher yields via leverage.

The Base Account is further designed to allow users to integrate automation to manage one's yield positions. It can be programmed by an individual to boost current leverage or deleverage or even close one's positions based on preconfigured triggers.

### 6.1.2 Using Registry Instead of Proxies

**Proxy Contracts** Proxy Contracts (not to confuse with the above proxy) is a design principle where the contract implementation and the contract state are split. The implementation contracts hold the business logic, and the Proxy Contract owns the state. This allows the contract deployer to change the business logic while keeping the state intact. But the individual users should direct the contracts always via the Proxy Contract.

**Registry** The registry is a design principle where the contract maintains the current implementation logic and keeps track of it via a Registry contract. The registry contract does not handle the state, unlike the Proxy.

We opted for the Registry design since most of our contracts do not maintain any state as a design choice and only facilitate users' state management via Base Account.

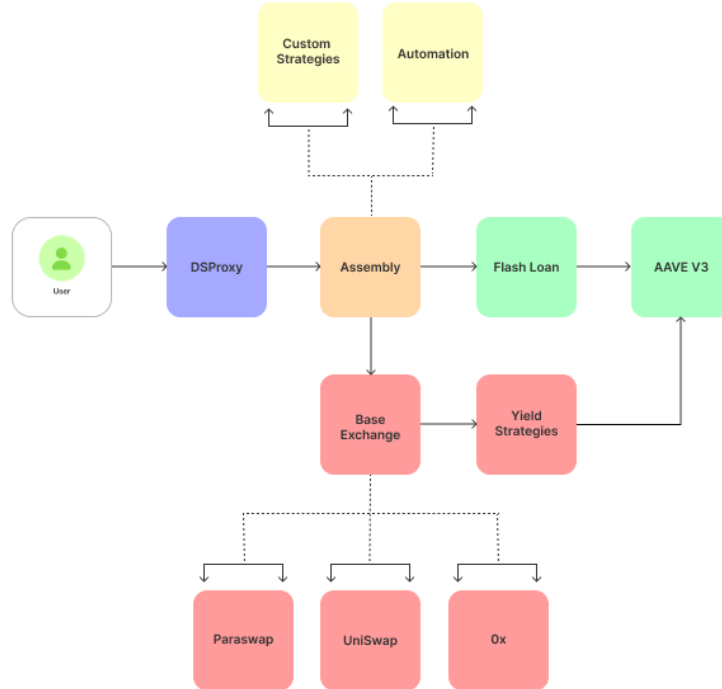


Figure 2: User interacts with DSPProxy and Assembly to bundle and execute DeFi transactions

## 7 Security

### 7.1 Security By Best Practices

#### 7.1.1 Static Analysis - Slither

Slither is a static analysis framework specifically designed for analyzing Solidity smart contracts on the Ethereum blockchain. It is a useful tool for developers, auditors, and security researchers to identify potential security vulnerabilities, design flaws, and other issues in their Solidity code before deploying it on the blockchain.

**Benefits of Slither:**

- **Comprehensive Analysis:** Slither performs a wide range of checks and analyses on Solidity contracts, including detecting common vulnerabilities like reentrancy bugs, integer overflow/underflow, uninitialized state variables, and more.
- **Reporting and Visualization:** The tool provides detailed reports and visualizations to help users understand the issues found in their contracts better.
- **Up-to-date:** The developers of Slither actively maintain and update the tool to keep up with the latest security best practices and vulnerabilities in the Solidity ecosystem.

Using Slither for analysis of Solidity contracts is beneficial because it helps identify potential security flaws and vulnerabilities that might otherwise go unnoticed during manual code reviews. Solidity smart contracts are immutable once deployed, so it is crucial to perform thorough analysis and testing before deployment to avoid any potential exploits or loss of funds. Slither can provide an additional layer of security assurance, especially for complex contracts, and is a valuable tool in the development and auditing process of Ethereum smart contracts.

## 7.2 Base Account

The project had 100+ issues spread across 83 detectors (classes of issues) to begin with. The issues were analyzed, fixed, and false positives were removed and triaged to finally arrive at 2 issues.

```

Reentrancy in ProxyRegistry.build(address) (contracts/ProxyRegistry.sol#25-30):
  External calls:
  - proxy = DSProxy(factory.build(owner)) (contracts/ProxyRegistry.sol#28)
  State variables written after the call(s):
  - proxies[owner] = proxy (contracts/ProxyRegistry.sol#29)
  ProxyRegistry.proxies (contracts/ProxyRegistry.sol#9) can be used in cross function reentrancies:
  - ProxyRegistry.build(address) (contracts/ProxyRegistry.sol#25-30)
  - ProxyRegistry.proxies (contracts/ProxyRegistry.sol#9)
Reference: https://github.com/crytic/slither/wiki/Detector-Documentation#reentrancy-vulnerabilities-1

Reentrancy in DSProxy.execute(bytes,bytes) (contracts/Proxy.sol#25-37):
  External calls:
  - target = cache.write(code) (contracts/Proxy.sol#33)
  Event emitted after the call(s):
  - LogNote(msg.sig,msg.sender,foo,bar,wad,msg.data) (contracts/ds-note/note.sol#27)
    - response = execute(target,data) (contracts/Proxy.sol#36)
Reference: https://github.com/crytic/slither/wiki/Detector-Documentation#reentrancy-vulnerabilities-3
. analyzed (21 contracts with 81 detectors), 2 result(s) found

```

Figure 3: Issue Analysis

**Issue 1:** The ProxyRegistry takes the address input to build an instance of DSProxy, and the passed instance becomes the owner of the DSProxy. Also, it is



created only once for the particular address until the current owner changes the owner. And proxies is the readonly call to get the proxy associated with the user. Hence we will mark this issue as a false positive.

**Issue 2:** The reentrancy is benign because its exploitation would have the same effect as two consecutive calls. This is what enables the bundling of transactions and hence this is a no fix.

**Link to the initial report:** [https://github.com/Sharpelabs/base-wallet/blob/forReview/slither\\_report\\_old.json](https://github.com/Sharpelabs/base-wallet/blob/forReview/slither_report_old.json)

**Link to the final report:** [https://github.com/Sharpelabs/base-wallet/blob/forReview/slither\\_report.json](https://github.com/Sharpelabs/base-wallet/blob/forReview/slither_report.json)

## 7.3 Base Platform

The project had 1000+ issues spread across 83 detectors (classes of issues) to begin with. The issues were analyzed, fixed, and false positives were removed and triaged to finally arrive at 5 issues.

```
FeeReceiver.withdrawEth(address,uint256)._to (contracts/utls/FeeReceiver.sol#39) lacks a zero-check on :  
- (success) = _to.call{value: _amount}() (contracts/utls/FeeReceiver.sol#44)
```

Figure 4: Issue 1: Fixed - The issue is about performing an input validation of the `_to` address to ensure ETH is not sent to an invalid address (ZERO Address).

```
FLAAveV3.executeOperation(address[],uint256[],uint256[],address,bytes) (contracts/actions/flashloan/FLAAveV3.sol#93-146) ignores return value by IDSPProxy(proxy).execute(value: address(this).balance)(assemblyExecutor,abi.encodeWithSignature(_executeDefiTxFromFL((string,bytes[],bytes32[],bytes4[],uint8[])),bytes32),currAssembly,bytes32(_amounts[0] + _fees[0])) (contracts/actions/flashloan/FLAAveV3.sol#118-125)  
FLAAction.executeOperation(address[],uint256[],uint256[],address,bytes) (contracts/actions/flashloan/FLAAction.sol#105-156) ignores return value by IDSPProxy(proxy).execute(value: address(this).balance)(recipeExecutor,abi.encodeWithSelector(CALLBACK_SELECTOR,currRecipe,bytes32(_amounts[0] + _fees[0])) (contracts/actions/flashloan/FLAAction.sol#130-133)  
AssemblyExecutor._parseFLAndExecute(AssemblyModel.Assembly,address,bytes32[]) (contracts/core/AssemblyExecutor.sol#92-117) ignores return value by DefiTxBase(_fLActionAddr).executeDefiTx(_currAssembly.callData[0],_currAssembly.subData,_currAssembly.paramMapping[0],_returnValues) (contracts/core/AssemblyExecutor.sol#109-114)  
Reference: https://github.com/crytic/slither/wiki/Detector-Documentation#unused-return
```

Figure 5: Issue 2, 3, 4: False Positives, Triaged - These set of issues have a common behavior that the `DSPProxy.execute` is called but its return response is not used. It is to be noted that the `DSPProxy` from Base Account is a generic call to bundle the DeFi transactions. In the case of success, the response can be varied and hence it is not possible to use its response. Events are used to process and show required data on the screen. While in the case of failure, `DSPProxy` reverts and the errors can be handled appropriately.

```
AssemblyExecutor.executeDefiTx(AssemblyModel.Assembly,uint256,bytes32[]) (contracts/core/AssemblyExecutor.sol#67-85) has external calls inside a loop: response = IDSPProxy(address(this)).execute(actionAddr,abi.encodeWithSignature(executeDefiTx(bytes,bytes32[],uint8[],bytes32[]),_currAssembly.callData[_index],_currAssembly.subData,_currAssembly.paramMapping[_index],_returnValues)) (contracts/core/AssemblyExecutor.sol#75-84)
Reference: https://github.com/crytic/slither/wiki/Detector-Documentation#calls-inside-a-loop
```

Figure 6: Issue 5: False Positive, Triaged - The issue is about using loops to bundle transactions as failure in one of the transactions can lead to failure of the whole bundle of transactions and thereby causing a denial of service. However, bundling of transactions is the inherent behavior of AssemblyExecutor. It accepts bundled transaction data and fwd it to DSPProxy so that the user can execute them.

## 7.4 Code Coverage - Solidity Coverage

Code coverage tools, such as solidity-coverage, play a crucial role in ensuring the quality and reliability of smart contracts in Solidity projects. They help developers and auditors assess the effectiveness of their test suites and identify areas of code that are not adequately tested. Here are some key reasons why code coverage tools are important in Solidity projects:

**Test Suite Effectiveness** Code coverage tools measure how much of the codebase is exercised during testing. By analyzing code coverage reports, developers can identify parts of their smart contracts that are not adequately covered by test cases. This information helps improve the test suite’s effectiveness by adding tests to cover those missed areas.

**Bug Detection** Higher code coverage indicates that more parts of the code have been tested, reducing the likelihood of bugs or vulnerabilities going unnoticed. It enhances the overall robustness of the smart contract and minimizes the risk of potential exploits.

**Security Assurance** Solidity code often deals with financial transactions and valuable assets. Ensuring high code coverage helps ensure that the contract behaves as intended, minimizing security risks and potential loss of funds.

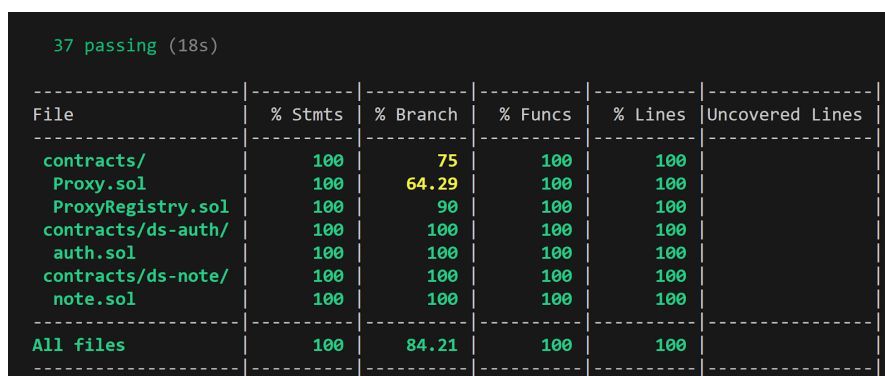
**Documentation and Code Understanding** Code coverage reports can serve as documentation for developers and auditors, helping them understand how different parts of the contract interact and highlighting potential areas of complexity or risk.

**Compliance** In some cases, regulatory requirements or security standards may mandate a certain level of code coverage for smart contracts. Code coverage tools help demonstrate compliance with these requirements.

Specifically, solidity-coverage is a popular code coverage tool for Solidity projects. It instruments the contract's bytecode to determine which parts of the code have been executed during test runs. It generates a detailed report showing the coverage percentage and highlights the lines of code that were and were not executed during testing. Solidity-coverage is essential for improving the quality, security, and reliability of Solidity smart contracts. They help developers identify areas of improvement in their test suites and provide confidence in the correctness and behavior of the deployed contracts.

### 7.4.1 Base Account

Solidity Coverage report for Base Account:



File	% Stmts	% Branch	% Funcs	% Lines	Uncovered Lines
contracts/Proxy.sol	100	75	100	100	
contracts/ProxyRegistry.sol	100	90	100	100	
contracts/ds-auth/auth.sol	100	100	100	100	
contracts/ds-note/note.sol	100	100	100	100	
All files	100	84.21	100	100	

Figure 7: Solidity Coverage Report for Base Account

## 8 Peer Code Reviews

Peer code review is a software development practice where developers review each other's code to identify defects, improve code quality, and ensure adherence to coding standards and best practices. It is a crucial part of the software development process, as it helps catch bugs, improve code maintainability, and foster knowledge sharing within the development team.

### 8.1 Purpose

The primary purpose of peer code review is to ensure that the code being introduced into the codebase is of high quality and meets the required standards. It aims to identify and fix defects, improve code readability, maintainability, and performance, and ensure the code aligns with the project's overall architecture and design.

## 8.2 Process

Peer code review typically involves the author submitting their code changes (a pull request) to be reviewed by one or more other developers (the reviewers). Reviewers provide feedback, suggestions, and comments on the code, and the author makes necessary changes based on the feedback.

## 8.3 Code Review Tool

There are various tools available that facilitate code reviews; we use GitHub pull requests.

## 8.4 Benefits

- **Bug Detection:** Reviewers can identify defects or logic errors that the original developer might have missed.
- **Knowledge Sharing:** Code reviews provide an opportunity for team members to learn from each other and gain a deeper understanding of the code base.
- **Code Consistency:** Code reviews ensure that the code adheres to coding standards and maintains consistency throughout the project.
- **Continuous Improvement:** Code reviews promote a culture of continuous improvement, helping developers grow and develop their skills.

## 9 Conclusion

This security report on Sharpe Base has provided a comprehensive analysis of various components, methodologies, and security measures within the platform. Key findings include the successful utilization of static analysis tools like Slither, code coverage with solidity coverage, and a structured approach to issue analysis and mitigation. The peer review process further ensures code consistency and continuous improvement. While some areas of risk were identified and addressed, the overall design and architecture of Sharpe Base demonstrate robustness and alignment with industry best practices. The report underscores Sharpe Labs' commitment to quality, security, and innovation, and provides actionable insights for continuous enhancement of the platform.