Zellic

safETH

Smart Contract Security Assessment

July 6, 2023

Prepared for:

Asymmetry Finance

Prepared by:

Ulrich Myhre and Sina Pilehchiha

Zellic Inc.

Contents

About Zellic

Zellic was founded in 2020 by a team of blockchain specialists with more than a decade of combined industry experience. We are leading experts in smart contracts and Web3 development, cryptography, web security, and reverse engineering. Before Zellic, we founded perfect blue, the top competitive hacking team in the world. Since then, our team has won countless cybersecurity contests and blockchain security events.

Zellic aims to treat clients on a case-by-case basis and to consider their individual, unique concerns and business needs. Our goal is to see the long-term success of our partners rather than simply provide a list of present security issues. Similarly, we strive to adapt to our partners' timelines and to be as available as possible. To keep up with our latest endeavors and research, check out our website zellic.io or follow @zellic_io on Twitter. If you are interested in partnering with Zellic, please contact us at hello@zellic.io.

1 Executive Summary

Zellic conducted a security assessment for Asymmetry Finance from June 26th to July 3rd, 2023. During this engagement, Zellic reviewed safETH's code for security vulnerabilities, design issues, and general weaknesses in security posture.

1.1 Goals of the Assessment

In a security assessment, goals are framed in terms of questions that we wish to answer. These questions are agreed upon through close communication between Zellic and the client. In this assessment, we sought to answer the following questions:

- Are users able to withdraw more than they are entitled?
- Are users able to properly stake and unstake without risking loss of their funds?
- Can deposits or withdrawals be manipulated to unfairly reward or penalize users?
- How do centralization risks impact the security and integrity of user funds?

1.2 Non-goals and Limitations

We did not assess the following areas that were outside the scope of this engagement:

- Problems relating to the front-end components and infrastructure of the project
- Problems due to improper key custody or off-chain access control
- Issues stemming from code or infrastructure outside of the assessment scope

Due to the time-boxed nature of security assessments in general, there are limitations in the coverage an assessment can provide.

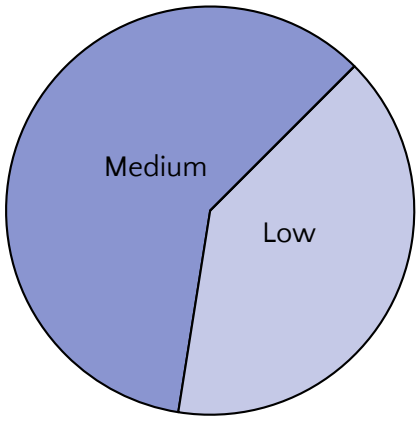
1.3 Results

During our assessment on the scoped safETH contracts, we discovered five findings. No critical issues were found. Three were of medium impact and two were of low impact.

Additionally, Zellic recorded its notes and observations from the assessment for Asymmetry Finance's benefit in the Discussion section (4) at the end of the document.

Breakdown of Finding Impacts

Impact Level	Count
Critical	0
High	0
Medium	3
Low	2
Informational	0



2 Introduction

2.1 About Asymmetry Finance — safETH

Asymmetry Finance is a DeFi protocol designed to decentralize the liquid-staked Ethereum derivative market through the LSTfi product suite led by safETH and afETH. Constructed as an index of liquid-staked Ethereum tokens (LSTs), Asymmetry offers users market-leading yield while diffusing the risk of holding just one LST through an index approach complete with the most efficient gas routing on the market, with zero added fees. Through afETH and safETH, users earn market-leading yield while making the Ethereum ecosystem more secure by driving decentralization forward for all.

2.2 Methodology

During a security assessment, Zellic works through standard phases of security auditing including both automated testing and manual review. These processes can vary significantly per engagement, but the majority of the time is spent on a thorough manual review of the entire scope.

Alongside a variety of tools and analyzers used on an as-needed basis, Zellic focuses primarily on the following classes of security and reliability issues:

Basic coding mistakes. Many critical vulnerabilities in the past have been caused by simple, surface-level mistakes that could have easily been caught ahead of time by code review. Depending on the engagement, we may also employ sophisticated analyzers such as model checkers, theorem provers, fuzzers, and so on as necessary. We also perform a cursory review of the code to familiarize ourselves with the contracts.

Business logic errors. Business logic is the heart of any smart contract application. We examine the specifications and designs for inconsistencies, flaws, and weaknesses that create opportunities for abuse. For example, these include problems like unrealistic tokenomics or dangerous arbitrage opportunities. To the best of our abilities, time permitting, we also review the contract logic to ensure that the code implements the expected functionality as specified in the platform's design documents.

Integration risks. Several well-known exploits have not been the result of any bug within the contract itself; rather, they are an unintended consequence of the contract's interaction with the broader DeFi ecosystem. Time permitting, we review the contracts' external interactions and summarize the associated risks: for example,

flash loan attacks, oracle price manipulation, MEV/sandwich attacks, and so on.

Code maturity. We look for potential improvements in the code base in general. We look for violations of industry best practices and guidelines and code quality standards. We also provide suggestions for possible optimizations, such as gas optimization, upgradeability weaknesses, centralization risks, and so on.

For each finding, Zellic assigns it an impact rating based on its severity and likelihood. There is no hard-and-fast formula for calculating a finding's impact. Instead, we assign it on a case-by-case basis based on our judgment and experience. Both the severity and likelihood of an issue affect its impact. For instance, a highly severe issue's impact may be attenuated by a low likelihood. We assign the following impact ratings (ordered by importance): Critical, High, Medium, Low, and Informational.

Zellic organizes its reports such that the most important findings come first in the document, rather than being strictly ordered on impact alone. Thus, we may sometimes emphasize an "Informational" finding higher than a "Low" finding. The key distinction is that although certain findings may have the same impact rating, their *importance* may differ. This varies based on various soft factors, like our clients' threat models, their business needs, and so on. We aim to provide useful and actionable advice to our partners considering their long-term goals, rather than a simple list of security issues at present.

2.3 Scope

The engagement involved a review of the following targets:

safETH Contracts

Repository	https://github.com/asymmetryfinance/smart-contracts
Version	smart-contracts: `eb0d83f95c95f5dd3135043988e20de1aa371ded`
Programs	• SafEth.sol • SafEthStorage.sol • Ankr.sol • DerivativeBase.sol • Reth.sol • SfrxEth.sol • Stafi.sol

- Swell.sol
- WstEth.sol

Type	Solidity
Platform	EVM-compatible

2.4 Project Overview

Zellic was contracted to perform a security assessment with two consultants for a total of nine person-days. The assessment was conducted over the course of one calendar week.

Contact Information

The following project manager was associated with the engagement:

Chad McDonald, Engagement Manager
chad@zellic.io

The following consultants were engaged to conduct the assessment:

Ulrich Myhre, Engineer
unblvr@zellic.io

Sina Pilehchiha, Engineer
sina@zellic.io

2.5 Project Timeline

The key dates of the engagement are detailed below.

June 26, 2023	Kick-off call
June 26, 2023	Start of primary review period
July 3, 2023	End of primary review period

3 Detailed Findings

3.1 Reentrancy in `withdrawEth`

- **Target**: SafEth
- **Category**: Coding Mistakes
- **Likelihood**: Low
- **Severity**: Medium
- **Impact**: Medium

Description

SafEth supports preminting, where the owner of the contract can stake some ETH and create a supply of SafEth. Staking into this supply is significantly cheaper (in gas) than exchanging every derivative. Whenever a user tries to stake an amount less than what is preminted, the input ETH will go to the SafEth contract and the user is instead given SafEth from the preminted supply, and the contract's `ethToClaim` will increase. The contract owner has access to a function that withdraws the ETH used to premint SafEth.

```
function withdrawEth() external onlyOwner {
    // solhint-disable-next-line
    (bool sent, ) = address(msg.sender).call{value: ethToClaim}("");
    if (!sent) revert FailedToSend();
    ethToClaim = 0;
}
```

This function lacks a reentrancy check and only resets the `ethToClaim` when the function ends. If the owner is compromised and replaced with a contract that reenters on payment, it is possible to extract all the ETH residing in the contract. However, the only current way to add ETH directly to the SafEth contract is through the premint staking mechanism. From the code pattern of tracking `ethToClaim`, it is clear that the intention is not to withdraw all the ETH in the contract through this function.

Impact

A compromised owner can empty the ETH balance of SafEth. Currently this is less of a problem because ETH rarely resides in the contract outside of the intended mechanism. However, the fix is easy and blocks future upgrades of the contract from being drained if it stores ETH.

Recommendations

We recommend modifying the function to comply with the checks-effects-interactions pattern,

```
function withdrawEth() external onlyOwner {
    uint256 _ethToClaim = ethToClaim;
    ethToClaim = 0;
    // solhint-disable-next-line
    (bool sent, ) = address(msg.sender).call{value: _ethToClaim}("");
    if (!sent) revert FailedToSend();
}
```

or add a reentrancy guard to the function.

Remediation

This issue has been acknowledged by Asymmetry Finance, and a fix was implemented in commit dc7b9c8e.

3.2 Function `doMultiStake()` does not spend all input ETH

- **Target**: SafEth
- **Category**: Coding Mistakes
- **Likelihood**: Medium
- **Severity**: Medium
- **Impact**: Medium

Description

When a user wants to stake more than `singleDerivativeThreshold`, and there is not enough left in the premint supply, the contract ends up calling `doMultiStake()`, which does a weighted stake into multiple derivatives.

```
...
    uint256 amountStaked = 0;
    for (uint256 i = 0; i < derivativeCount; i++) {
        if (!derivatives[i].enabled) continue;
        uint256 weight = derivatives[i].weight;
        if (weight == 0) continue;
        IDerivative derivative = derivatives[i].derivative;
        uint256 ethAmount = i == derivativeCount - 1
            ? msg.value - amountStaked
            : (msg.value * weight) / totalWeight;

        amountStaked += ethAmount;
         ...
    }
...
```

This portion of the code shows that it is iterating over all the derivatives, skipping the disabled ones. For each derivative, it stakes `(msg.value * weight) / totalWeight` ETH, which rounds down slightly due to integer division.

To account for the rounding issue, the last iteration stakes `msg.value - amountStaked`, where the latter is the accumulated value of staked ETH so far. If the last derivative is disabled, the last iteration is skipped due to `if (!derivatives[i].enabled) continue;`, and the rounding is not accounted for.

Impact

When the last derivative is disabled, and depending on the actual weights and the staked amount, a small percentage of the staked amount can be left in the contract. This will not be caught by the derivative slippage checks, but it can be caught by the

user's `_minOut` parameter when properly set. Disabling the last derivative in the list thus leads to either a loss of funds for the user or blocking the functionality of staking into multiple derivatives from working.

Recommendations

The protocol should not rely on the last derivative to be enabled. A possible fix could be to implement something like a `getLastEnabledDerivativeIndex()` instead, which returns the real index of the last derivative, replacing `derivativeCount -1`. This can also reduce the amount of iterations ran by the for loop.

Remediation

This issue has been acknowledged by Asymmetry Finance, and a fix was implemented in commit e4a2864e.

3.3 Function `firstUnderweightDerivativeIndex()` returns a valid index on error

- **Target**: SafEth
- **Category**: Coding Mistakes
- **Likelihood**: Medium
- **Severity**: Medium
- **Impact**: Medium

Description

When a single stake is triggered, the SafEth contract tries to find the best target derivative to stake into by calling `firstUnderweightDerivativeIndex()`.

```
function doSingleStake(
        uint256 _minOut,
        uint256 price
    ) private returns (uint256 mintedAmount) {
        uint256 totalStakeValueEth = 0;
        IDerivative derivative
    = derivatives[firstUnderweightDerivativeIndex()]
            .derivative;
        uint256 depositAmount = derivative.deposit{value: msg.value}();
         ...
    }
...

function firstUnderweightDerivativeIndex() private view returns (uint256)
    {
    uint256 count = derivativeCount;

    uint256 tvlEth = totalSupply() * approxPrice(false);

    if (tvlEth == 0) return 0;

    for (uint256 i = 0; i < count; i++) {
        if (!derivatives[i].enabled) continue;
        uint256 trueWeight = (totalWeight *
            IDerivative(derivatives[i].derivative).balance() *
            IDerivative(derivatives[i].derivative).ethPerDerivative(
                false
            )) / tvlEth;
        if (trueWeight < derivatives[i].weight) return i;
```

```
        }
        return 0;
    }
```

The function iterates over all the enabled derivatives, calculating a "true weight" by calculating `(totalWeight * derivative_balance_value_in_ETH) / safEth_value_in_ETH`. If this value is less than the derivative weight, it is considered underweight and has its index returned. Disabled derivatives are not considered, and their non-contribution is already accounted for in `totalWeight`.

If the total supply of SafEth is 0, or none of the derivatives are considered underweight, a default value of `0` is returned. This index is then used in `doSingleStake` without checking if that derivative is disabled.

Impact

If none of the derivatives are underweight, or the total supply of SafEth is 0, a single stake can end up staking into a disabled derivative. The functionality of disabling derivatives is used in the cases when they appear to be more centralized or get depegged or corrupted somehow. Depending on the reason for disabling the derivative, the impact can vary greatly, from total loss of funds to just giving business to a derivative that is getting too centralized.

Recommendations

There are many proper ways to fix this. One example could be this: for both `tvlEth == 0` and the default return, fall back to finding the first nondisabled derivative. Revert if there are none.

Remediation

This issue has been acknowledged by Asymmetry Finance, and a fix was implemented in commit 4247587b.

3.4 Floor price is reset on consecutive premints

- **Target**: SafEth
- **Category**: Coding Mistakes
- **Likelihood**: Low
- **Severity**: Medium
- **Impact**: Low

Description

When the contract owner premints SafEth, the `depositPrice` value returned from calling `stake()` is saved. It is stored in the global value `floorPrice` and acts as a minimum price to be paid when using the preminted supply during staking. Thanks to the `floorPrice`, the owners will be able to recoup their investment even if the price should go down later.

```
function preMint(
    uint256 _minAmount,
    bool _useBalance
) external payable onlyOwner returns (uint256) {
    uint256 amount = msg.value;
    ...
    (uint256 mintedAmount, uint256 depositPrice) = this.stake{
        value: amount
    }(_minAmount);

    floorPrice = depositPrice;
    ...
}

function shouldPremint(uint256 price) private view returns (bool) {
    uint256 preMintPrice = price < floorPrice ? floorPrice : price;
    uint256 amount = (msg.value * 1e18) / preMintPrice;
    return amount ≤ preMintedSupply && msg.value ≤ maxPreMintAmount;
}
```

In the `preMint()` function, `floorPrice` is set directly and will overwrite the previous value there.

Impact

If a premint is executed during a time when the price is high, `floorPrice` will be set to that high price. If another premint happens before the previous supply is depleted,

the `floorPrice` will be reset to the new `depositPrice`. If the price changed, this will under or overvalue the remaining preminted supply. The owner will then risk losing parts of the ETH invested during preminting, as it gets valued at a lower price than it was traded at.

If the premint supply is (more or less) depleted before minting again, the problem can be avoided.

Recommendations

There is no easy way to tie a certain price to just a part of the premint supply. A possibility could be to introduce a parameter `bool reset_floorprice` to `preMint()`, which allows the `floorPrice` to be reduced. Otherwise, it is limited to only increase, or it remains unchanged (but is checked to be within some limit).

Remediation

This issue has been acknowledged by Asymmetry Finance, and a fix was implemented in commit ac8ae472.

3.5 Preminting can stake from premint supply

- **Target**: SafEth
- **Category**: Coding Mistakes
- **Likelihood**: Low
- **Severity**: Low
- **Impact**: Low

Description

When the owner calls `preMint()`, the function ends up calling `this.stake()`, the same function that external users can call.

```
function stake(
    uint256 _minOut
)
    external
    payable
    nonReentrant
    returns (uint256 mintedAmount, uint256 depositPrice)
{
    depositPrice = approxPrice(true);
    if (shouldPremint(depositPrice))
        return doPreMintedStake(_minOut, depositPrice);
    if (msg.value < singleDerivativeThreshold)
        return (doSingleStake(_minOut, depositPrice), depositPrice);
    return (doMultiStake(_minOut, depositPrice), depositPrice);
}
```

The `stake()` function calculates the `depositPrice` and decides if the current price and premint supply can cover the amount being staked. If it can, the caller will be given SafEth from the preminted supply instead of minting new tokens and doing the full exchange of every derivative. This saves a significant amount of gas.

An issue appears if the owner calls `preMint()` with an amount that can be covered by the premint supply. Then the preminter will be given already preminted tokens, which is unexpected.

Impact

It is impossible to premint an amount of tokens smaller than what is already in the premint supply. If it is smaller, no new SafEth will be minted, but preminted supply will be recycled into preminted supply.

Recommendations

Add a `require` in `preMint()` that checks if the preminting amount is less than the premint supply.

Remediation

This issue has been acknowledged by Asymmetry Finance, and a fix was implemented in commit 24320f40.

4 Discussion

The purpose of this section is to document miscellaneous observations that we made during the assessment. These discussion notes are not necessarily security related and do not convey that we are suggesting a code change.

4.1 Centralization risks

In the rare event where an owner — or other privileged entity — is compromised, the possible consequences should be outlined for transparency.

SafEth

The SafEth contract has an ownership model where the owner has access to several privileged functions:

- `setBlacklistedRecipient`
- `setWhitelistedSender`
- `setSingleDerivativeThreshold`
- `preMint`
- `withdrawEth`
- `derivativeRebalance`
- `adjustWeight`
- `disableDerivative`
- `enableDerivative`
- `addDerivative`
- `setPauseStaking`
- `setPauseUnstaking`
- Ownership transfer (two-step) and renouncing
- Various getters and setters, for example threshold values

While none of these alone can lead to theft of customer funds, there are some dangerous compositions. Given a compromised owner, it is possible to call `addDerivative()` to add an attacker-controlled contract, followed by several `derivativeRebalance()` calls to convert back all the existing derivatives to ETH, then deposit them into the attacker-controlled derivative. Luckily, this is hard to do by accident and would require a total compromise of the `owner` role inside the SafEth contract.

Asymmetry Finance states that the owner is a multi-sig, which should adequately

protect against a single malicious user, but there are no timelocks or other safeguards that let the user react to changes like the one outlined above.

Even the `withdrawEth()` function, if called by accident, phishing, or other type of trickery (that does not totally compromise the user), will just withdraw the accumulated ETH that the owner has preminted. No customer funds are at risk by calling this (given the remediation for finding 3.1). It also goes back to `msg.sender` and not an arbitrary account.

Renouncing ownership is a dangerous function that *can* be called by accident. It is recommended that the function for this is overridden if the contract is supposed to always have an owner. The risk here would be that ownership is renounced while, for example, unstaking is paused. This could potentially lock away staked funds forever. The same is true for situations when access to the owner account is lost, for example, through hardware malfunction, theft, natural disasters, or death.

Pausing staking, unstaking, or disabling all derivatives also opens up an opportunity for denial of service or taking ransoms for releasing customer funds.

Derivatives

The derivatives Ankr, Reth, SfrxEth, Stafi, Swell, and WstEth are all based on DerivativeBase. Derivatives specify two roles: `owner` and `manager`. The `owner` is set to be the deployed SafEth contract, and there is no code inside that contract that can change the owner of a derivative (yet — however, the contract is upgradeable).

The manager has access to only a handful of privileged functions:

- `setMaxSlippage`
- `setChainlinkFeed` (only in Reth and WstEth)

A compromise of this account would, for most derivatives, only give the ability to affect the final bookkeeping checks in `finalChecks()`. That function tries to verify that the received amount is not outside the slippage tolerance set in the contract. Being able to set the Chainlink feed is riskier, as it allows someone to set up a fake Chainlink contract that gives much better rates to a certain `tx.origin`.

Loss of access to the manager account would not be as dramatic as for the SafEth contract. Not being able to tweak the slippage tolerance could disallow raising of the maximum staking limit, as `maxSlippage` is not a percentage of the amount but hard-coded as, for example, `0.01 ETH`.

5 Threat Model

This provides a full threat model description for various functions. As time permitted, we analyzed each function in the smart contracts and created a written threat model for some critical functions. A threat model documents a given function's externally controllable inputs and how an attacker could leverage each input to cause harm.

Not all functions in the audit scope may have been modeled. The absence of a threat model in this section does not necessarily suggest that a function is safe.

5.1 Module: Ankr.sol

Function: `deposit()`

Deposits into derivative.

Branches and code coverage (including function calls)

Intended branches

- Should properly deposit ETH into derivative if called by the owner.
  ☑ Test coverage

Negative behavior

- Should only allow the owner to call this function.
  ☐ Negative test

Function call analysis

- `rootFunction → AnkrStaker.stakeAndClaimAethC()`
  - **What is controllable?** N/A.
  - **If return value controllable, how is it used and how can it go wrong?** Discarded.
  - **What happens if it reverts, reenters, or does other unusual control flow?** Discarded.
- `rootFunction → super.finalChecks(uint256, uint256, uint256, uint256, bool, uint256)`
  - **What is controllable?** N/A.
  - **If return value controllable, how is it used and how can it go wrong?** Discarded.

- **What happens if it reverts, reenters, or does other unusual control flow?** Discarded.

Function: `initialize(address _owner)`

Initialize values for the contracts.

Inputs

- `_owner`
  - **Control**: Full.
  - **Constraints**: Must be a valid address.
  - **Impact**: The owner of the contract will be set.

Branches and code coverage (including function calls)

Intended branches

- Should ensure that address `_owner` is valid.
  ☐ Test coverage

Negative behavior

- If `_owner` is not a valid address, function will revert.
  ☐ Negative test

Function call analysis

- `rootFunction → DerivativeBase.init(address)`
  - **What is controllable?** `_owner` address.
  - **If return value controllable, how is it used and how can it go wrong?** N/A.
  - **What happens if it reverts, reenters, or does other unusual control flow?** Discarded.

Function: `setMaxSlippage(uint256 _slippage)`

Sets max slippage for derivative.

Inputs

- `_slippage`
  - **Control**: Fully controlled by manager.
  - **Constraints**: N/A.
  - **Impact**: Determines the maximum slippage.

Branches and code coverage (including function calls)

Intended branches

- Determines maximum slippage.
  ☑ Test coverage

Negative behavior

- If not called by a `manager` role, the function will revert.
  ☐ Negative test

Function: `withdraw(uint256 _amount)`

Converts derivative into ETH.

Inputs

- `_amount`
  - **Control**: Fully controlled.
  - **Constraints**: N/A.
  - **Impact**: Determines the amount to be converted.

Branches and code coverage (including function calls)

Intended branches

- When the caller has `onlyOwner` role, `_amount` derivative is converted into ETH.
  ☑ Test coverage

Negative behavior

- If the caller is not the owner, the function should revert.
  ☑ Negative test

Function call analysis

- `rootFunction → IERC20.approve(address, uint256)`
  - **What is controllable?** `_amount`.
  - **If return value controllable, how is it used and how can it go wrong?** Discarded.
  - **What happens if it reverts, reenters, or does other unusual control flow?** Discarded.
- `rootFunction → IAnkrEthEthPool.exchange(int128, int128, uint256, uint256)`

  - **What is controllable?** `_amount`.
  - **If return value controllable, how is it used and how can it go wrong?** Discarded.
  - **What happens if it reverts, reenters, or does other unusual control flow?** Discarded.
- `rootFunction → DerivativeBase.finalChecks(uint256, uint256, uint256, uint256, bool, uint256)`
  - **What is controllable?** `_amount`.
  - **If return value controllable, how is it used and how can it go wrong?** Discarded.
  - **What happens if it reverts, reenters, or does other unusual control flow?** Discarded.

5.2 Module: DerivativeBase.sol

Function: `initializeV2()`

Sets the manager address for the derivative.

Branches and code coverage (including function calls)

Intended branches

- Properly sets the manager address if called for the first time.
  ☑ Test coverage

Negative behavior

- Should not allow calling `initializeV2` after the first time.
  ☑ Negative test

Function: `updateManager(address _manager)`

Updates manager address.

Inputs

- `_manager`
  - **Control**: Fully controlled.
  - **Constraints**: Should be nonzero.
  - **Impact**: Sets the new manager address.

Branches and code coverage (including function calls)

Intended branches

- Properly sets the new manager address.
  ☑ Test coverage

Negative behavior

- Should fail when called by nonmanager.
  ☑ Negative test
- Should fail when the new address is zero.
  ☑ Negative test

5.3 Module: Reth.sol

Function: `deposit()`

Deposits into derivative.

Branches and code coverage (including function calls)

Intended branches

- Should properly deposit ETH into derivative if called by owner.
  ☑ Test coverage

Negative behavior

- Should only allow owner to call this function.
  ☐ Negative test

Function call analysis

- `rootFunction → balancerSwap(uint256)`
  - **What is controllable?** `msg.value`.
  - **If return value controllable, how is it used and how can it go wrong?** Discarded.
  - **What happens if it reverts, reenters, or does other unusual control flow?** Discarded.
- `rootFunction → DerivativeBase.finalChecks(uint256, uint256, uint256, uint256, bool, uint256)`
  - **What is controllable?** N/A.
  - **If return value controllable, how is it used and how can it go wrong?** Dis-

carded.
  - **What happens if it reverts, reenters, or does other unusual control flow?** Discarded.

Function: `initialize(address _owner)`

Initialize values for the contracts.

Inputs

- `_owner`
  - **Control**: Full.
  - **Constraints**: Must be a valid address.
  - **Impact**: The owner of the contract will be set.

Branches and code coverage (including function calls)

Intended branches

- Should ensure that address `_owner` is valid.
  - ☐ Test coverage

Negative behavior

- If `_owner` is not a valid address, function will revert.
  - ☐ Negative test

Function call analysis

- `rootFunction → DerivativeBase.init(address)`
  - **What is controllable?** `_owner` address.
  - **If return value controllable, how is it used and how can it go wrong?** N/A.
  - **What happens if it reverts, reenters, or does other unusual control flow?** Discarded.

Function: `setChainlinkFeed(address _priceFeedAddress)`

Sets the address for the Chainlink feed.

Inputs

- `_priceFeedAddress`
  - **Control**: Fully controlled by `manager` role.
  - **Constraints**: N/A.

- **Impact**: Determines the address for the Chainlink feed.

Branches and code coverage (including function calls)

Intended branches

- Manager determines the address for the Chainlink feed.
  ☐ Test coverage

Function: `setMaxSlippage(uint256 _slippage)`

Sets max slippage for derivative.

Inputs

- `_slippage`
  - **Control**: Fully controlled by manager.
  - **Constraints**: N/A.
  - **Impact**: Determines the maximum slippage.

Branches and code coverage (including function calls)

Intended branches

- Determines maximum slippage.
  ☑ Test coverage

Negative behavior

- If not called by a `manager` role, the function will revert.
  ☐ Negative test

Function: `withdraw(uint256 _amount)`

Converts derivative into ETH.

Inputs

- `_amount`
  - **Control**: Fully controlled.
  - **Constraints**: N/A.
  - **Impact**: Determines the amount to be converted.

Branches and code coverage (including function calls)

Intended branches

- When the caller has `onlyOwner` role, the `_amount` derivative is converted into ETH.
  ☑ Test coverage

Negative behavior

- If the caller is not the owner, the function should revert.
  ☑ Negative test

Function call analysis

- `rootFunction → ethPerDerivative(bool)`
  - **What is controllable?** N/A.
  - **If return value controllable, how is it used and how can it go wrong?** Discarded.
  - **What happens if it reverts, reenters, or does other unusual control flow?** Discarded.
- `rootFunction → IERC20.approve(address, uint256)`
  - **What is controllable?** `_amount`.
  - **If return value controllable, how is it used and how can it go wrong?** Discarded.
  - **What happens if it reverts, reenters, or does other unusual control flow?** Discarded.
- `rootFunction → RocketSwapRouterInterface.swapFrom(uint256, uint256, uint256, uint256, uint256)`
  - **What is controllable?** `_amount`.
  - **If return value controllable, how is it used and how can it go wrong?** Discarded.
  - **What happens if it reverts, reenters, or does other unusual control flow?** Discarded.
- `rootFunction → IWETH.withdraw(uint256)`
  - **What is controllable?** N/A.
  - **If return value controllable, how is it used and how can it go wrong?** Discarded.
  - **What happens if it reverts, reenters, or does other unusual control flow?** Discarded.
- `rootFunction → DerivativeBase.finalChecks(uint256, uint256, uint256, uint256, bool, uint256)`
  - **What is controllable?** `_amount`.

- **If return value controllable, how is it used and how can it go wrong?** Discarded.
- **What happens if it reverts, reenters, or does other unusual control flow?** Discarded.

5.4 Module: SafEth.sol

Function: `addDerivative(address _contractAddress, uint256 _weight)`

Adds new derivative to the index fund.

Inputs

- `_contractAddress`
  - **Control**: Full.
  - **Constraints**: Should be a valid and supported contract address.
  - **Impact**: Address of the derivative contract launched by AF.
- `_weight`
  - **Control**: Full.
  - **Constraints**: N/A.
  - **Impact**: New weight for this derivative.

Branches and code coverage (including function calls)

Intended branches

- Properly adds new derivative to the index fund.
  ☑ Test coverage

Negative behavior

- Should fail with adding non-ERC-165 compliant derivative.
  ☑ Negative test
- Should fail with adding invalid ERC-165 derivative.
  ☑ Negative test

Function call analysis

- `rootFunction → ERC165.supportsInterface(bytes4)`
  - **What is controllable?** `_contractAddress`.
  - **If return value controllable, how is it used and how can it go wrong?** Discarded.

  - **What happens if it reverts, reenters, or does other unusual control flow?** Discarded.
- `rootFunction → setTotalWeight()`
  - **What is controllable?** N/A.
  - **If return value controllable, how is it used and how can it go wrong?** Discarded.
  - **What happens if it reverts, reenters, or does other unusual control flow?** Discarded.

Function: `adjustWeight(uint256 _derivativeIndex, uint256 _weight)`

Changes derivative weight based on derivative index.

Inputs

- `_derivativeIndex`
  - **Control**: Full.
  - **Constraints**: Needs to be greater than or equal to `derivativeCount`.
  - **Impact**: Index of the derivative you want to update the weight.
- `_weight`
  - **Control**: Full.
  - **Constraints**: N/A.
  - **Impact**: New weight for this derivative.

Branches and code coverage (including function calls)

Intended branches

- Should properly change weights and rebalance.
  ☑ Test coverage

Negative behavior

- Should revert if `totalWeight` is 0.
  ☑ Negative test
- Should revert if called by nonowner.
  ☐ Negative test

Function call analysis

- `rootFunction → setTotalWeight()`
  - **What is controllable?** N/A.
  - **If return value controllable, how is it used and how can it go wrong?** Dis-

carded.
  - **What happens if it reverts, reenters, or does other unusual control flow?** Discarded.

Function: `derivativeRebalance(uint256 _sellDerivativeIndex, uint256 _buyDerivativeIndex, uint256 _sellAmount)`

Allows owner to rebalance between two derivatives, selling one for the other.

Inputs

- `_sellDerivativeIndex`
  - **Control**: Full.
  - **Constraints**: Needs to be greater or equal to `derivativeCount`. If equal to `_buyDerivativeIndex`, it reverts.
  - **Impact**: Index of the derivative to sell.
- `_buyDerivativeIndex`
  - **Control**: Full.
  - **Constraints**: Needs to be greater than or equal to `derivativeCount`. If equal to `_sellDerivativeIndex`, it reverts.
  - **Impact**: Index of the derivative to buy.
- `_sellAmount`
  - **Control**: Full.
  - **Constraints**: If equal to 0, revert with `AmountTooLow()`.
  - **Impact**: Amount of the derivative to sell.

Branches and code coverage (including function calls)

Intended branches

- Properly rebalances between two derivatives.
  ☑ Test coverage

Negative behavior

- Should revert if called by nonowner.
  ☐ Negative test

Function call analysis

- `rootFunction → withdraw(uint256)`
  - **What is controllable?** `_sellAmount`.
  - **If return value controllable, how is it used and how can it go wrong?** Dis-

carded.
  - **What happens if it reverts, reenters, or does other unusual control flow?** Discarded.
- `rootFunction → deposit()`
  - **What is controllable?** N/A.
  - **If return value controllable, how is it used and how can it go wrong?** Discarded.
  - **What happens if it reverts, reenters, or does other unusual control flow?** Discarded.

Function: `disableDerivative(uint256 _derivativeIndex)`

Disables derivative based on derivative index.

Inputs

- `_derivativeIndex`
  - **Control**: Full.
  - **Constraints**: Needs to be greater than or equal to `derivativeCount` and enabled.
  - **Impact**: Index of the derivative you want to disable.

Branches and code coverage (including function calls)

Intended branches

- Properly disables derivative when called by owner role.
  ☑ Test coverage

Negative behavior

- Should revert if called by nonowner.
  ☑ Negative test
- Should fail to disable a nonexistent derivative.
  ☑ Negative test
- Should fail to disable an already disabled derivative.
  ☑ Negative test

Function call analysis

- `rootFunction → setTotalWeight()`
  - **What is controllable?** N/A.
  - **If return value controllable, how is it used and how can it go wrong?** Dis-

carded.
  - **What happens if it reverts, reenters, or does other unusual control flow?** Discarded.

Function: `enableDerivative(uint256 _derivativeIndex)`

Enables derivative based on derivative index.

Inputs

- `_derivativeIndex`
  - **Control**: Full.
  - **Constraints**: Needs to be greater than or equal to `derivativeCount` and enabled.
  - **Impact**: Index of the derivative you want to enable.

Branches and code coverage (including function calls)

Intended branches

- Properly enables derivative when called by `owner` role.
  ☑ Test coverage

Negative behaviour

- Should revert if called by nonowner.
  ☑ Negative test
- Should fail to enable a nonexistent derivative.
  ☑ Negative test
- Should fail to enable an already enabled derivative.
  ☑ Negative test

Function call analysis

- `rootFunction → setTotalWeight()`
  - **What is controllable?** N/A.
  - **If return value controllable, how is it used and how can it go wrong?** Discarded.
  - **What happens if it reverts, reenters, or does other unusual control flow?** Discarded.

Function: `initialize(string _tokenName, string _tokenSymbol)`

Initializes values for the contracts.

Inputs

- `_tokenName`
  - **Control**: Full.
  - **Constraints**: N/A.
  - **Impact**: Name of ERC-20.
- `_tokenSymbol`
  - **Control**: Full.
  - **Constraints**: N/A.
  - **Impact**: Symbol of ERC-20.

Branches and code coverage (including function calls)

Intended branches

- Properly initializes values for the contracts.
  - ☐ Test coverage

Negative behavior

- Should not allow calling after first time.
  - ☐ Negative test

Function: `preMint(uint256 _minAmount, bool _useBalance)`

Premints safEth for future users.

Inputs

- `_minAmount`
  - **Control**: Full.
  - **Constraints**: N/A.
  - **Impact**: Minimum amount to stake.
- `_useBalance`
  - **Control**: Full.
  - **Constraints**: N/A.
  - **Impact**: Should use balance from previous premint's to mint more.

Branches and code coverage (including function calls)

Intended branches

- User should be able to call `preMint()`, passing `_useBalance` as true.
  ☑ Test coverage

Negative behavior

- Cannot premint if not owner.
  ☑ Negative test

Function call analysis

- `rootFunction → stake(uint256)`
  - **What is controllable?** `_minAmount`.
  - **If return value controllable, how is it used and how can it go wrong?** Discarded.
  - **What happens if it reverts, reenters, or does other unusual control flow?** Discarded.

Function: `receive()`

The `receive()` function.

Branches and code coverage (including function calls)

Intended branches

- Properly allows ETH being sent from derivative contracts.
  ☑ Test coverage

Negative behavior

- Should revert if sent ETH by a user.
  ☑ Negative test

Function: `setBlacklistedRecipient(address _recipient, bool _isBlacklisted)`

Sets a recipient address as blacklisted to receive tokens.

Inputs

- `_recipient`
  - **Control**: Full.

  - **Constraints**: N/A.
  - **Impact**: Recipient address to set blacklist on/off.
- `_isBlacklisted`
  - **Control**: Full.
  - **Constraints**: N/A.
  - **Impact**: True or false.

Branches and code coverage (including function calls)

Intended branches

- Should allow owner to edit the whitelist and blacklist.
  ☑ Test coverage

Negative behavior

- Should fail if nonowner tries to edit the blacklist.
  ☐ Negative test

Function: `setMaxAmount(uint256 _maxAmount)`

Sets the maximum amount a user is allowed to stake.

Inputs

- `_maxAmount`
  - **Control**: Full.
  - **Constraints**: N/A.
  - **Impact**: amount to set as maximum stake value.

Branches and code coverage (including function calls)

Intended branches

- Should be able to change min/max.
  ☑ Test coverage

Negative behavior

- Should only allow owner to call min/max functions.
  ☑ Negative test

Function: `setMaxPreMintAmount(uint256 _amount)`

Sets the maximum amount a user can premint in one transaction

Inputs

- `_minAmount`
  - **Control**: Full.
  - **Constraints**: N/A.
  - **Impact**: Amount to set as maximum premint value

Branches and code coverage (including function calls)

Intended branches

- Should change max premint amount.
  ☑ Test coverage

Negative behavior

- Can't change max premint if not owner
  ☑ Negative test

Function: `setMinAmount(uint256 _minAmount)`

Sets the minimum amount a user is allowed to stake.

Inputs

- `_minAmount`
  - **Control**: Full.
  - **Constraints**: N/A.
  - **Impact**: Amount to set as minimum stake value.

Branches and code coverage (including function calls)

Intended branches

- Should be able to change min/max.
  ☑ Test coverage

Negative behavior

- Should only allow owner to call min/max functions.
  ☑ Negative test

Function: `setPauseStaking(bool _pause)`

Enables/disables the stake function.

Inputs

- `_pause`
  - **Control**: Full.
  - **Constraints**: N/A.
  - **Impact**: True disables staking / false enables staking.

Branches and code coverage (including function calls)

Intended branches

- Properly enables/disables the stake function.
  ☑ Test coverage

Negative behavior

- Should fail to call `setPauseStaking()` if setting the same value.
  ☑ Negative test
- Should only allow the owner to call the function.
  ☑ Negative test

Function: `setPauseUnstaking(bool _pause)`

Enables/disables the unstake function.

Inputs

- `_pause`
  - **Control**: Full.
  - **Constraints**: N/A.
  - **Impact**: True disables unstaking / false enables unstaking.

Branches and code coverage (including function calls)

Intended branches

- Properly enables/disables the unstake function.
  ☑ Test coverage

Negative behavior

- Should fail to call `setPauseUnstaking()` if setting the same value.
  ☑ Negative test
- Should fail to call `setPauseUnstaking()` if called by the nonowner.
  ☑ Negative test

Function: `setSingleDerivativeThreshold(uint256 _amount)`

Sets the ETH amount at which it will use standard weighting versus buying a single derivative.

Inputs

- `_amount`
  - **Control**: Full.
  - **Constraints**: N/A.
  - **Impact**: Amount of ETH where it will switch to standard weighting.

Branches and code coverage (including function calls)

Intended branches

- Should properly test `setSingleDerivativeThreshold()`.
  ☑ Test coverage
- Unchecked.
  ☐ Test coverage

Negative behavior

- Should fail if nonowner tries to call the function.
  ☐ Negative test

Function: `setWhitelistedSender(address _sender, bool _isWhitelisted)`

Sets a sender address as whitelisted to send to blacklisted addresses.

Inputs

- `_sender`
  - **Control**: Full.
  - **Constraints**: N/A.
  - **Impact**: Sender address to set whitelist on/off.
- `_isWhitelisted`
  - **Control**: Full.
  - **Constraints**: N/A.
  - **Impact**: True or false.

Branches and code coverage (including function calls)

Intended branches

- Should allow owner to edit the whitelist.
  ☑ Test coverage

Negative behavior

- Should fail if nonowner tries to edit the whitelist and blacklist.
  ☐ Negative test

Function: `stake(uint256 _minOut)`

Stakes ETH into safETH.

Inputs

- `_minOut`
  - **Control**: Full.
  - **Constraints**: Discarded.
  - **Impact**: Minimum amount of safETH to mint.

Branches and code coverage (including function calls)

Intended branches

- Properly stakes ETH into safETH.
  ☑ Test coverage

Negative behavior

- Should fail with wrong min/max.
  ☑ Negative test
- Should fail staking through `preMint` with `minOut` higher than expected SafEth output.
  ☑ Negative test

Function call analysis

- `rootFunction → approxPrice(bool)`
  - **What is controllable?** N/A.
  - **If return value controllable, how is it used and how can it go wrong?** Discarded.
  - **What happens if it reverts, reenters, or does other unusual control flow?** Discarded.
- `rootFunction → shouldPremint(uint256)`
  - **What is controllable?** N/A.

  - **If return value controllable, how is it used and how can it go wrong?** Discarded.
  - **What happens if it reverts, reenters, or does other unusual control flow?** Discarded.
- `rootFunction → doPreMintedStake(uint256, uint256)`
  - **What is controllable?** `_minOut`.
  - **If return value controllable, how is it used and how can it go wrong?** Discarded.
  - **What happens if it reverts, reenters, or does other unusual control flow?** Discarded.
- `rootFunction → doSingleStake(uint256, uint256)`
  - **What is controllable?** `_minOut`.
  - **If return value controllable, how is it used and how can it go wrong?** Discarded.
  - **What happens if it reverts, reenters, or does other unusual control flow?** Discarded.
- `rootFunction → doMultiStake(uint256, uint256)`
  - **What is controllable?** `_minOut`.
  - **If return value controllable, how is it used and how can it go wrong?** Discarded.
  - **What happens if it reverts, reenters, or does other unusual control flow?** Discarded.

Function: `transferFrom(address sender, address recipient, uint256 amount)`

Standard ERC-20 `transferFrom()` with `checkBlacklist` modifier.

Inputs

- `sender`
  - **Control**: Full.
  - **Constraints**: `checkBlacklist(sender, recipient)`.
  - **Impact**: `sender` address.
- `recipient`
  - **Control**: Full.
  - **Constraints**: `checkBlacklist(sender, recipient)`.
  - **Impact**: `recipient` address.
- `amount`
  - **Control**: Full.

  - **Constraints**: Discarded.
  - **Impact**: Amount to be transferred.

Branches and code coverage (including function calls)

Intended branches

- Should successfully `transferFrom()`, even if recipient is blacklisted.
  ☑ Test coverage

Negative behavior

- Should fail `transferFrom()` to blacklisted address from a non-whitelisted address.
  ☑ Negative test

Function call analysis

- `rootFunction → ERC20Upgradeable.transferFrom(address, address, uint256)`
  - **What is controllable?** `sender`, `recipient`, and `amount`.
  - **If return value controllable, how is it used and how can it go wrong?** Discarded.
  - **What happens if it reverts, reenters, or does other unusual control flow?** Discarded.

Function: `transfer(address _recipient, uint256 _amount)`

Standard ERC-20 `transfer()` with `checkBlacklist` modifier.

Inputs

- `_recipient`
  - **Control**: Full.
  - **Constraints**: `checkBlacklist(sender, recipient)`.
  - **Impact**: `_recipient` address.
- `_amount`
  - **Control**: Full.
  - **Constraints**: `checkBlacklist(sender, recipient)`.
  - **Impact**: `_amount` to transfer.

Branches and code coverage (including function calls)

Intended branches

- Should successfully `transferFrom()`, even if recipient is blacklisted.
  ☑ Test coverage

Negative behavior

- Should fail `transfer()` to blacklisted address from a non-whitelisted address.
  ☑ Negative test

Function call analysis

- `rootFunction → transfer(address, uint256)`
  - **What is controllable?** `_recipient` and `_amount`.
  - **If return value controllable, how is it used and how can it go wrong?** Discarded.
  - **What happens if it reverts, reenters, or does other unusual control flow?** Discarded.

Function: `unstake(uint256 _safEthAmount, uint256 _minOut)`

Unstakes safETH into ETH.

Inputs

- `_safEthAmount`
  - **Control**: Full.
  - **Constraints**: Cannot be equal to 0 and needs to be sufficiently large, otherwise `InsufficientBalance()`.
  - **Impact**: Amount of safETH to unstake into ETH.
- `_minOut`
  - **Control**: Full.
  - **Constraints**: N/A.
  - **Impact**: Minimum amount of ETH to unstake.

Branches and code coverage (including function calls)

Intended branches

- Properly unstakes safETH.
  ☑ Test coverage

Negative behavior

- Should fail unstake on zero `safEthAmount`.
  ☑ Negative test

- Should fail unstake on invalid `safEthAmount`.
  ☑ Negative test
- Should revert if reentering unstake.
  ☑ Negative test

Function call analysis

- `rootFunction → totalSupply()`
  - **What is controllable?** N/A.
  - **If return value controllable, how is it used and how can it go wrong?** Discarded.
  - **What happens if it reverts, reenters, or does other unusual control flow?** Discarded.
- `rootFunction → withdraw(uint256)`
  - **What is controllable?** N/A.
  - **If return value controllable, how is it used and how can it go wrong?** Discarded.
  - **What happens if it reverts, reenters, or does other unusual control flow?** Discarded.

Function: `withdrawEth()`

Claims ETH that was used to acquire preminted safEth.

Branches and code coverage (including function calls)

Intended branches

- Properly claims ETH.
  ☑ Test coverage

Negative behavior

- Cannot claim funds if not the owner.
  ☑ Negative test

5.5 Module: SfrxEth.sol

Function: `deposit()`

Deposits into derivative.

Branches and code coverage (including function calls)

Intended branches

- Should properly deposit ETH into derivative if called by owner.
  ☑ Test coverage

Negative behavior

- Should only allow owner to call this function.
  ☐ Negative test

Function call analysis

- `rootFunction → frxETHMinterContract.submitAndDeposit(address)`
  - **What is controllable?** N/A.
  - **If return value controllable, how is it used and how can it go wrong?** Discarded.
  - **What happens if it reverts, reenters, or does other unusual control flow?** Discarded.
- `rootFunction → DerivativeBase.finalChecks(uint256, uint256, uint256, uint256, bool, uint256)`
  - **What is controllable?** `msg.value`.
  - **If return value controllable, how is it used and how can it go wrong?** Discarded.
  - **What happens if it reverts, reenters, or does other unusual control flow?** Discarded.

Function: `initialize(address _owner)`

Initialize values for the contracts.

Inputs

- `_owner`
  - **Control**: Full.
  - **Constraints**: Must be a valid address.
  - **Impact**: The owner of the contract will be set.

Branches and code coverage (including function calls)

Intended branches

- Should ensure that address `_owner` is valid.

☐ Test coverage

Negative behavior

- If `_owner` is not a valid address, function will revert.
  ☐ Negative test

Function call analysis

- `rootFunction → DerivativeBase.init(address)`
  - **What is controllable?** `_owner` address.
  - **If return value controllable, how is it used and how can it go wrong?** N/A.
  - **What happens if it reverts, reenters, or does other unusual control flow?** Discarded.

Function: `setDepegSlippage(uint256 _depegSlippage)`

Set depeg slippage.

Inputs

- `_depegSlippage`
  - **Control**: Full.
  - **Constraints**: N/A.
  - **Impact**: Slippage amount to revert at.

Branches and code coverage (including function calls)

Intended branches

- Should `setDepegSlippage()` on SfrxEth derivative.
  ☑ Test coverage

Function: `setMaxSlippage(uint256 _slippage)`

Sets max slippage for derivative.

Inputs

- `_slippage`
  - **Control**: Fully controlled by manager.
  - **Constraints**: N/A.
  - **Impact**: Determines the maximum slippage.

Branches and code coverage (including function calls)

Intended branches

- Determines maximum slippage.
  ☑ Test coverage

Negative behavior

- If not called by a `manager` role, the function will revert.
  ☐ Negative test

Function: `withdraw(uint256 _amount)`

Converts derivative into ETH.

Inputs

- `_amount`
  - **Control**: Fully controlled by owner.
  - **Constraints**: N/A.
  - **Impact**: Determines amount to be converted.

Branches and code coverage (including function calls)

Intended branches

- When the caller has `onlyOwner` role, the `_amount` derivative is converted into ETH.
  ☑ Test coverage

Negative behavior

- If the caller is not the owner, the function should revert.
  ☑ Negative test

Function call analysis

- `rootFunction → IsFrxEth.redeem(uint256, address, address)`
  - **What is controllable?** `_amount_`.
  - **If return value controllable, how is it used and how can it go wrong?** Discarded.
  - **What happens if it reverts, reenters, or does other unusual control flow?** Discarded.
- `rootFunction → IsFrxEth.approve(address, uint256)`
  - **What is controllable?** N/A.

  - **If return value controllable, how is it used and how can it go wrong?** Discarded.
  - **What happens if it reverts, reenters, or does other unusual control flow?** Discarded.
- `rootFunction → IsFrxEth.exchange(int128, int128, uint256, uint256)`
  - **What is controllable?** N/A.
  - **If return value controllable, how is it used and how can it go wrong?** Discarded.
  - **What happens if it reverts, reenters, or does other unusual control flow?** Discarded.
- `rootFunction → DerivativeBase.finalChecks(uint256, uint256, uint256, uint256, bool, uint256)`
  - **What is controllable?** `_amount`.
  - **If return value controllable, how is it used and how can it go wrong?** Discarded.
  - **What happens if it reverts, reenters, or does other unusual control flow?** Discarded.

5.6 Module: Stafi.sol

Function: `deposit()`

Deposits into Stafi derivative.

Branches and code coverage (including function calls)

Intended branches

- Should properly deposit ETH into derivative if called by owner.
  ☑ Test coverage

Negative behavior

- Should only allow owner to call this function.
  ☐ Negative test

Function call analysis

- `rootFunction → IWETH.deposit()`
  - **What is controllable?** N/A.
  - **If return value controllable, how is it used and how can it go wrong?** Discarded.

  - **What happens if it reverts, reenters, or does other unusual control flow?** Discarded.
- `rootFunction → IERC20.approve(address, uint256)`
  - **What is controllable?** N/A.
  - **If return value controllable, how is it used and how can it go wrong?** Discarded.
  - **What happens if it reverts, reenters, or does other unusual control flow?** Discarded.
- `rootFunction → BALANCER_VAULT.swap(SingleSwap memory, FundManagement memory, uint256, uint256)`
  - **What is controllable?** N/A.
  - **If return value controllable, how is it used and how can it go wrong?** Discarded.
  - **What happens if it reverts, reenters, or does other unusual control flow?** Discarded.
- `rootFunction → DerivativeBase.finalChecks(uint256, uint256, uint256, uint256, bool, uint256)`
  - **What is controllable?** `msg.value`.
  - **If return value controllable, how is it used and how can it go wrong?** Discarded.
  - **What happens if it reverts, reenters, or does other unusual control flow?** Discarded.

Function: `initialize(address _owner)`

Initialize values for the contracts.

Inputs

- `_owner`
  - **Control**: Full.
  - **Constraints**: Must be a valid address.
  - **Impact**: The owner of the contract will be set.

Branches and code coverage (including function calls)

Intended branches

- Should ensure that address `_owner` is valid.
  - ☐ Test coverage

Negative behavior

- If `_owner` is not a valid address, function will revert.
  ☐ Negative test

Function call analysis

- `rootFunction → DerivativeBase.init(address)`
  - **What is controllable?** `_owner` address.
  - **If return value controllable, how is it used and how can it go wrong?** N/A.
  - **What happens if it reverts, reenters, or does other unusual control flow?** Discarded.

Function: `setMaxSlippage(uint256 _slippage)`

Sets max slippage for derivative.

Inputs

- `_slippage`
  - **Control**: Fully controlled by manager.
  - **Constraints**: N/A.
  - **Impact**: Determines the maximum slippage.

Branches and code coverage (including function calls)

Intended branches

- Determines maximum slippage.
  ☑ Test coverage

Negative behavior

- If not called by a manager role, the function will revert.
  ☐ Negative test

Function: `withdraw(uint256 _amount)`

Converts derivative into ETH.

Inputs

- `_amount`
  - **Control**: Fully controlled by owner.
  - **Constraints**: N/A.
  - **Impact**: Determines amount to be converted.

Branches and code coverage (including function calls)

Intended branches

- When the caller has `onlyOwner` role, the `_amount` derivative is converted into ETH.
  ☑ Test coverage

Negative behavior

- If the caller is not the owner, the function should revert.
  ☑ Negative test

Function call analysis

- `rootFunction → IERC20.approve(address, uint256)`
  - **What is controllable?** `_amount`.
  - **If return value controllable, how is it used and how can it go wrong?** Discarded.
  - **What happens if it reverts, reenters, or does other unusual control flow?** Discarded.
- `rootFunction → BALANCER_VAULT.swap(SingleSwap memory, FundManagement memory, uint256, uint256)`
  - **What is controllable?** N/A.
  - **If return value controllable, how is it used and how can it go wrong?** Discarded.
  - **What happens if it reverts, reenters, or does other unusual control flow?** Discarded.
- `rootFunction → DerivativeBase.finalChecks(uint256, uint256, uint256, uint256, bool, uint256)`
  - **What is controllable?** `_amount`.
  - **If return value controllable, how is it used and how can it go wrong?** Discarded.
  - **What happens if it reverts, reenters, or does other unusual control flow?** Discarded.

5.7 Module: Swell.sol

Function: `deposit()`

Deposits into SWETH derivative.

Branches and code coverage (including function calls)

Intended branches

- Should properly deposit ETH into derivative if called by owner.
  ☑ Test coverage

Negative behavior

- Should only allow owner to call this function.
  ☐ Negative test

Function call analysis

- `rootFunction → swapInputSingle(uint256, address, address)`
  - **What is controllable?** `_amount_`.
  - **If return value controllable, how is it used and how can it go wrong?** Discarded.
  - **What happens if it reverts, reenters, or does other unusual control flow?** Discarded.
- `rootFunction → IWETH.deposit()`
  - **What is controllable?** N/A.
  - **If return value controllable, how is it used and how can it go wrong?** Discarded.
  - **What happens if it reverts, reenters, or does other unusual control flow?** Discarded.
- `rootFunction → DerivativeBase.finalChecks(uint256, uint256, uint256, uint256, bool, uint256)`
  - **What is controllable?** N/A.
  - **If return value controllable, how is it used and how can it go wrong?** Discarded.
  - **What happens if it reverts, reenters, or does other unusual control flow?** Discarded.

Function: `initialize(address _owner)`

Initialize values for the contracts.

Inputs

- `_owner`
  - **Control**: Full.
  - **Constraints**: Must be a valid address.

- **Impact**: The owner of the contract will be set.

Branches and code coverage (including function calls)

Intended branches

- Should ensure that address `_owner` is valid.
  ☐ Test coverage

Negative behavior

- If `_owner` is not a valid address, function will revert.
  ☐ Negative test

Function call analysis

- `rootFunction → DerivativeBase.init(address)`
  - **What is controllable?** `_owner` address.
  - **If return value controllable, how is it used and how can it go wrong?** N/A.
  - **What happens if it reverts, reenters, or does other unusual control flow?** Discarded.

Function: `setMaxSlippage(uint256 _slippage)`

Sets max slippage for derivative.

Inputs

- `_slippage`
  - **Control**: Fully controlled by manager.
  - **Constraints**: N/A.
  - **Impact**: Determines the maximum slippage.

Branches and code coverage (including function calls)

Intended branches

- Determines maximum slippage.
  ☑ Test coverage

Negative behavior

- If not called by a `manager` role, the function will revert.
  ☐ Negative test

Function: `withdraw(uint256 _amount)`

Converts derivative into ETH.

Inputs

- `_amount`
  - **Control**: Fully controlled by owner.
  - **Constraints**: N/A.
  - **Impact**: Determines amount to be converted.

Branches and code coverage (including function calls)

Intended branches

- When the caller has `onlyOwner` role, `_amount` derivative is converted into ETH.
  ☑ Test coverage

Negative behavior

- If the caller is not the owner, the function should revert.
  ☑ Negative test

Function call analysis

- `rootFunction → swapInputSingle(uint256, address, address)`
  - **What is controllable?** `_amount_`.
  - **If return value controllable, how is it used and how can it go wrong?** Discarded.
  - **What happens if it reverts, reenters, or does other unusual control flow?** Discarded.
- `rootFunction → IWETH.withdraw(uint256)`
  - **What is controllable?** N/A.
  - **If return value controllable, how is it used and how can it go wrong?** Discarded.
  - **What happens if it reverts, reenters, or does other unusual control flow?** Discarded.
- `rootFunction → DerivativeBase.finalChecks(uint256, uint256, uint256, uint256, bool, uint256)`
  - **What is controllable?** `_amount`.
  - **If return value controllable, how is it used and how can it go wrong?** Discarded.
  - **What happens if it reverts, reenters, or does other unusual control flow?**

Discarded.

5.8 Module: WstEth.sol

Function: `deposit()`

Deposits ETH into derivative.

Branches and code coverage (including function calls)

Intended branches

- Should properly deposit ETH into derivative if called by owner.
  ☑ Test coverage

Negative behavior

- Should only allow owner to call this function.
  ☐ Negative test

Function call analysis

- `rootFunction → DerivativeBase.finalChecks(uint256, uint256, uint256, uint256, bool, uint256)`
  - **What is controllable?** N/A.
  - **If return value controllable, how is it used and how can it go wrong?** Discarded.
  - **What happens if it reverts, reenters, or does other unusual control flow?** Discarded.

Function: `initialize(address _owner)`

Initialize values for the contracts.

Inputs

- `_owner`
  - **Control**: Full.
  - **Constraints**: Must be a valid address.
  - **Impact**: The owner of the contract will be set.

Branches and code coverage (including function calls)

Intended branches

- Should ensure that address `_owner` is valid.
  ☐ Test coverage

Negative behavior

- If `_owner` is not a valid address, function will revert.
  ☐ Negative test

Function call analysis

- `rootFunction → DerivativeBase.init(address)`
  - **What is controllable?** `_owner` address.
  - **If return value controllable, how is it used and how can it go wrong?** N/A.
  - **What happens if it reverts, reenters, or does other unusual control flow?** Discarded.

Function: `setChainlinkFeed(address _priceFeedAddress)`

Sets the address for the Chainlink feed.

Inputs

- `_priceFeedAddress`
  - **Control**: Fully controlled by `manager` role.
  - **Constraints**: N/A.
  - **Impact**: Determines the address for the Chainlink feed.

Branches and code coverage (including function calls)

Intended branches

- Manager determines the address for the Chainlink feed.
  ☐ Test coverage

Function: `setMaxSlippage(uint256 _slippage)`

Sets max slippage for derivative.

Inputs

- `_slippage`

  - **Control**: Fully controlled by the manager.
  - **Constraints**: N/A.
  - **Impact**: Determines the maximum slippage.

Branches and code coverage (including function calls)

Intended branches

- Determines maximum slippage.
  ☑ Test coverage

Negative behavior

- If not called by a `manager` role, the function will revert.
  ☐ Negative test

Function: `withdraw(uint256 _amount)`

Converts derivative into ETH.

Inputs

- `_amount`
  - **Control**: Fully controlled by owner.
  - **Constraints**: N/A.
  - **Impact**: Determines amount to be converted.

Branches and code coverage (including function calls)

Intended branches

- When the caller has `onlyOwner` role, `_amount` derivative is converted into ETH.
  ☑ Test coverage

Negative behavior

- If the caller is not the owner, the function should revert.
  ☑ Negative test

Function call analysis

- `rootFunction → IWStETH.unwrap(uint256)`
  - **What is controllable?** The `_amount` to withdraw.
  - **If return value controllable, how is it used and how can it go wrong?** Discarded.

  - **What happens if it reverts, reenters, or does other unusual control flow?** Discarded.
- `rootFunction → IERC20.approve(address, uint256)`
  - **What is controllable?** N/A.
  - **If return value controllable, how is it used and how can it go wrong?** Discarded.
  - **What happens if it reverts, reenters, or does other unusual control flow?** Discarded.
- `rootFunction → IStEthEthPool.exchange(int128, int128, uint256, uint256)`
  - **What is controllable?** N/A.
  - **If return value controllable, how is it used and how can it go wrong?** Discarded.
  - **What happens if it reverts, reenters, or does other unusual control flow?** Discarded.
- `rootFunction → DerivativeBase.finalChecks(uint256, uint256, uint256, uint256, bool, uint256)`
  - **What is controllable?** `_amount`.
  - **If return value controllable, how is it used and how can it go wrong?** Discarded.
  - **What happens if it reverts, reenters, or does other unusual control flow?** Discarded.

6 Audit Results

At the time of our audit, the audited code was not deployed to the Ethereum Mainnet.

During our assessment on the scoped safETH contracts, we discovered five findings. No critical issues were found. Three were of medium impact and two were of low impact. Asymmetry Finance acknowledged all findings and implemented fixes.

6.1 Disclaimer

This assessment does not provide any warranties about finding all possible issues within its scope; in other words, the evaluation results do not guarantee the absence of any subsequent issues. Zellic, of course, also cannot make guarantees about any code added to the project after the audit version of our assessment. Furthermore, because a single assessment can never be considered comprehensive, we always recommend multiple independent assessments paired with a bug bounty program.

For each finding, Zellic provides a recommended solution. All code samples in these recommendations are intended to convey how an issue may be resolved (i.e., the idea), but they may not be tested or functional code.

Finally, the contents of this assessment report are for informational purposes only; do not construe any information in this report as legal, tax, investment, or financial advice. Nothing contained in this report constitutes a solicitation or endorsement of a project by Zellic.