Zellic

Prepared for
Alex Qin
WOOFI

Prepared by
Ulrich Myhre
Vlad Toie
Zellic

August 16, 2024

WOOFI Stake

Smart Contract Security Assessment

Contents

About Zellic

Zellic is a vulnerability research firm with deep expertise in blockchain security. We specialize in EVM, Move (Aptos and Sui), and Solana as well as Cairo, NEAR, and Cosmos. We review L1s and L2s, cross-chain protocols, wallets and applied cryptography, zero-knowledge circuits, web applications, and more.

Prior to Zellic, we founded the #1 CTF (competitive hacking) team ↗ worldwide in 2020, 2021, and 2023. Our engineers bring a rich set of skills and backgrounds, including cryptography, web security, mobile security, low-level exploitation, and finance. Our background in traditional information security and competitive hacking has enabled us to consistently discover hidden vulnerabilities and develop novel security research, earning us the reputation as the go-to security firm for teams whose rate of innovation outpaces the existing security landscape.

For more on Zellic's ongoing security research initiatives, check out our website zellic.io ↗ and follow @zellic_io ↗ on Twitter. If you are interested in partnering with Zellic, contact us at hello@zellic.io ↗.

1. Overview

1.1. Executive Summary

Zellic conducted a security assessment for WOOFI from July 29th to August 7th, 2024. During this engagement, Zellic reviewed WOOFI Stake's code for security vulnerabilities, design issues, and general weaknesses in security posture.

1.2. 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:

- Can assets be lost in the smart contracts?
- Are the assets in the supercharger and staking vaults secure?
- Is cross-chain communication implemented securely?
- Is rewards calculation working as expected?

1.3. Non-goals and Limitations

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

- Front-end components
- Infrastructure relating to the project
- Key custody
- Parts of reward calculation that were out of scope

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

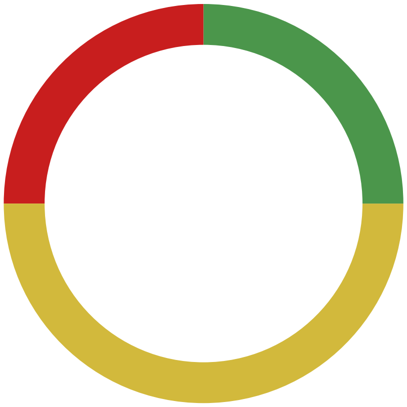
1.4. Results

During our assessment on the scoped WOOFI Stake contracts, we discovered four findings. One critical issue was found. Two were of medium impact and one was of low impact.

Additionally, Zellic recorded its notes and observations from the assessment for WOOFI's benefit in the Discussion section (4. ↗).

Breakdown of Finding Impacts

Impact Level	Count
🟥 Critical	1
🟧 High	0
🟨 Medium	2
🟩 Low	1
◼ Informational	0



2. Introduction

2.1. About WOOFI Stake

WOOFI contributed the following description of WOOFI Stake:

> https://learn.woo.org/
>
> WOOFi is a unique decentralized exchange that bridges the deep liquidity of centralized exchanges on chain. This enables DeFi traders to swap with size and maximize their profits through the lowest swap fees and minimal slippage.
>
> In addition to this, WOOFi has four other core components:
>
> ### Crosschain swaps
>
> - Move any asset quickly and seamlessly across 11 supported chains with minimum slippage
>
> ### Revenue sharing
>
> - Stake WOO tokens and earn 80% of all swap fees, paid in USDC or auto compound into WOO
>
> ### Supercharged yields
>
> - Lend assets to WOOFi's liquidity manager and earn leading single-sided yield, free of impermanent loss
>
> ### Perpetual futures
>
> - Trade perpetual futures with WOOFi Pro's order book, enjoy a CeFi style trading experience while keeping self-custody

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 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 codebase 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, upgradability 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.

Finally, Zellic provides a list of miscellaneous observations that do not have security impact or are not directly related to the scoped contracts itself. These observations — found in the Discussion (4. ↗) section of the document — may include suggestions for improving the codebase, or general recommendations, but do not necessarily convey that we suggest a code change.

2.3. Scope

The engagement involved a review of the following targets:

WOOFI Stake Contracts

Type	Solidity
Platform	EVM-compatible

Target: WooPoolV2

Repository	https://github.com/woonetwork/WooPoolV2 ↗
Version	`fb6be9d8f313ab935245133e9c1b1bcdf169cbf0`

Programs	`VaultV2.sol`
	`WooLendingManager.sol`
	`WooSuperChargerVaultV2.sol`
	`WooWithdrawManagerV2.sol`
	`StrategyAave.sol`
	`BaseStrategy.sol`

Target: WooStakingV2

Repository	https://github.com/woonetwork/WooStakingV2 ↗
Version	`248a72eee5845b570ae35e51d24b711e67348a93`

Programs	`WooStakingLocal.sol`
	`WooStakingController.sol`
	`WooStakingProxy.sol`
	`WooStakingManager.sol`
	`WooStakingCompounder.sol`

2.4. Project Overview

Zellic was contracted to perform a security assessment with two consultants for a total of 2.4 person-weeks. The assessment was conducted over the course of two calendar weeks.

Contact Information

The following project manager was associated with the engagement:

Jacob Goreski
Engagement Manager
jacob@zellic.io ↗

The following consultants were engaged to conduct the assessment:

Ulrich Myhre
Engineer
unblvr@zellic.io ↗

Vlad Toie
Engineer
vlad@zellic.io ↗

2.5. Project Timeline

The key dates of the engagement are detailed below.

July 29, 2024	Kick-off call
July 29, 2024	Start of primary review period
August 7, 2024	End of primary review period

3. Detailed Findings

3.1. ERC-1155 `mint` can be reentered with

Target	RewardCampaignManager		
Category	Business Logic	**Severity**	Critical
Likelihood	High	**Impact**	Critical

Description

We note that this particular finding addresses an issue with a contract that is outside of the scope of the audit.

The `mint` function of an ERC-1155 contract performs a so-called acceptance check on contracts that are receiving tokens.

For this check to be properly implemented, the contract that is receiving tokens must implement the `IERC1155Receiver` interface. This interface includes the function `onERC1155Received`, which is called by the `mint` function of the ERC-1155 contract. This procedure is potentially problematic, as malicious actors can implement custom `onERC1155Received` functions that reenter the calling sequence of the ERC-1155 contract.

In this case, the `_claim` function performs a `mint` call on the `rewardNFT` contract. This call can be reentered by a malicious actor, which can lead to a reentrancy attack, potentially minting multiple tokens of the same `tokenId` to the same user.

```
function _claim(uint256 _campaignId, address _user) internal returns (uint128)
    {
    require(isActiveCampaign[_campaignId], "RewardCampaignManager:
    !_campaignId");
    uint128 count = 0;
    uint256[] memory tokenIds = rewardNFT.getAllTokenIds();
    uint256 len = tokenIds.length;
    for (uint256 i = 0; i < len; ++i) {
        if (users[_campaignId][tokenIds[i]].contains(_user) &&
    !isClaimedUser[_campaignId][tokenIds[i]][_user]) {
            rewardNFT.mint(_user, tokenIds[i], 1);
            isClaimedUser[_campaignId][tokenIds[i]][_user] = true;
            count++;
        }
    }
    return count;
}
```

Impact

A malicious user can exploit this vulnerability to mint multiple tokens of the same `tokenId`, potentially leading to further issues involving the claimed tokens.

Recommendations

We recommend moving the `isClaimedUser` status update before the `mint` call to prevent reentrancy attacks.

```
function _claim(uint256 _campaignId, address _user) internal returns (uint128)
    {
    require(isActiveCampaign[_campaignId], "RewardCampaignManager:
    !_campaignId");
    uint128 count = 0;
    uint256[] memory tokenIds = rewardNFT.getAllTokenIds();
    uint256 len = tokenIds.length;
    for (uint256 i = 0; i < len; ++i) {
        if (users[_campaignId][tokenIds[i]].contains(_user) &&
    !isClaimedUser[_campaignId][tokenIds[i]][_user]) {
          rewardNFT.mint(_user, tokenIds[i], 1);
          isClaimedUser[_campaignId][tokenIds[i]][_user] = true;
          isClaimedUser[_campaignId][tokenIds[i]][_user] = true;
          rewardNFT.mint(_user, tokenIds[i], 1);
            count++;
        }
    }
    return count;
}
```

This way, the `isClaimedUser` status is updated before the `mint` call, preventing reentrancy attacks.

Remediation

This issue has been acknowledged by WOOFI, and a fix was implemented in commit 7d453754 ↗.

3.2. Possible fee leeching

Target	VaultV2		
Category	Business Logic	**Severity**	Medium
Likelihood	Low	**Impact**	Medium

Description

The `earn` function in VaultV2 takes the entire available balance of the want token and deposits it into the strategy, for accruing yield.

```
function earn() public override {
    if (_isStratActive()) {
        uint256 balanceAvail = available();
        TransferHelper.safeTransfer(want, address(strategy), balanceAvail);
        strategy.deposit();
    }
}
```

The deposit of all the available balance, however, might hinder the ability of users to withdraw want tokens from the vault. In order for a user withdrawal to occur, the Vault must have enough want tokens in the first place. This means that the Vault must withdraw additional tokens from the strategy in order to fulfill the user's withdrawal request.

```
function withdraw(uint256 shares) public override nonReentrant {
    // ...
    uint256 balanceBefore = IERC20(want).balanceOf(address(this));
    if (balanceBefore < withdrawAmount) {
        uint256 balanceToWithdraw = withdrawAmount - balanceBefore;
        require(_isStratActive(), "WOOFiVaultV2: STRAT_INACTIVE");
        strategy.withdraw(balanceToWithdraw);
        // ...
    }
}
```

In turn, this withdraw incurs additional fees, which are not accounted for in the Vault's accounting.

```
function withdraw(uint256 amount) public override nonReentrant {
    require(msg.sender == vault, "StrategyAave: !vault");
    require(amount > 0, "StrategyAave: !amount");
```

```
        uint256 wantBal = balanceOfWant();

        if (wantBal < amount) {
            IAavePool(aavePool).withdraw(want, amount - wantBal, address(this));
            uint256 newWantBal = IERC20(want).balanceOf(address(this));
            require(newWantBal > wantBal, "StrategyAave: !newWantBal");
            wantBal = newWantBal;
        }

        uint256 withdrawAmt = amount < wantBal ? amount : wantBal;

        uint256 fee = chargeWithdrawalFee(withdrawAmt);
        if (withdrawAmt > fee) {
            TransferHelper.safeTransfer(want, vault, withdrawAmt - fee);
        }
        emit Withdraw(balanceOf());
    }
```

Impact

A malicious user may purposefully withdraw small amounts of tokens, just enough that the Vault has to withdraw additional tokens from the strategy, incurring fees. This can be repeated multiple times, leading to a situation where the Vault is drained of its funds.

Recommendations

We recommend disallowing the call of `earn` for all but privileged users, such as the owner of the Vault. This way, the Vault can control the interval at which the `earn` function is called and can ensure there are enough funds in the Vault to cover user withdrawals.

Remediation

This issue has been acknowledged by WOOFI, and a fix was implemented in commit `651aae35` ↗.

3.3. Possible DOS

Target	WooSuperChargerVaultV2		
Category	Coding Mistakes	**Severity**	Medium
Likelihood	Medium	**Impact**	Medium

Description

The `requestWithdrawShares` is part of the withdrawal mechanism for the users of the SuperChargerVault:

```
function _requestWithdrawShares(uint256 shares) private {
    require(shares > 0, "WooSuperChargerVault: !amount");
    require(!isSettling, "WooSuperChargerVault: CANNOT_WITHDRAW_IN_SETTLING");
    require(requestUsers.length() <= maxWithdrawCount, "WooSuperChargerVault:
    MAX_WITHDRAW_COUNT");

    address owner = msg.sender;

    lendingManager.accureInterest();
    uint256 amount = _assets(shares);
    TransferHelper.safeTransferFrom(address(this), owner, address(this),
    shares);

    requestedWithdrawShares[owner] = requestedWithdrawShares[owner] + shares;
    requestedTotalShares = requestedTotalShares + shares;
    requestUsers.add(owner);

    emit RequestWithdraw(owner, amount, shares);
}
```

This function marks the amount of shares that the user wants to withdraw and adds the user to the list of users that have requested a withdrawal.

```
function endWeeklySettle() public onlyAdmin {
    require(isSettling, "!SETTLING");
    require(weeklyNeededAmountForWithdraw() == 0, "WEEKLY_REPAY_NOT_CLEARED");

    // NOTE: Do need accureInterest here, since it's already been updated in
    `startWeeklySettle`
    uint256 sharePrice = getPricePerFullShare();
    isSettling = false;
    uint256 totalWithdrawAmount = requestedTotalAmount();

    if (totalWithdrawAmount != 0) {
        uint256 shares = _sharesUp(totalWithdrawAmount,
    reserveVault.getPricePerFullShare());
        reserveVault.withdraw(shares);

        if (want == weth) {
            IWETH(weth).deposit{value:  totalWithdrawAmount}();
        }
        require(available() >= totalWithdrawAmount);

        uint256 length = requestUsers.length();

        // @audit-issue DOS here if attacker performs lots of requests.
        for (uint256 i = 0; i < length; i++) {
            address user = requestUsers.at(0);

            withdrawManager.addWithdrawAmount(user,
    (requestedWithdrawShares[user] * sharePrice) / 1e18);

            requestedWithdrawShares[user] = 0;
            requestUsers.remove(user);
        }

        _burn(address(this), requestedTotalShares);
        requestedTotalShares = 0;

        TransferHelper.safeTransfer(want, address(withdrawManager),
    totalWithdrawAmount);
    }

    instantWithdrawnAmount = 0;

    lendingManager.accureInterest();
    uint256 totalBalance = balance();
    instantWithdrawCap = totalBalance / 10;

    emit WeeklySettleEnded(msg.sender, totalBalance, lendingBalance(),
    reserveBalance());
}
```

The `endWeeklySettle` function is called by the admin to end the weekly settlement. It calculates the total amount of shares that users have requested to withdraw and then loops through the list of users that have requested a withdrawal. For each user, it calculates the amount of shares that the user has requested to withdraw and then adds this amount to the `withdrawManager`.

If a malicious user was to perform a large number of requests to withdraw shares, the `endWeeklySettle` function would have to loop through all of these requests, which could lead to a DOS attack, leaving the contract unusable.

Impact

The contract might become unusable, temporarily locking up funds.

Recommendations

The team is aware of this issue and has partly addressed it by limiting the number of withdrawal requests that can be made, through the `maxWithdrawCount` variable.

We recommend taking appropriate measures to prevent this DOS attack by constantly monitoring the gas consumption and limitations of the chain that the contract is deployed on.

Remediation

This issue has been acknowledged by WOOFI, and a fix was implemented in commit `da6cf56f` ↗.

A new function `batchEndWeeklySettle` was implemented in order to split the weekly settlement into batches. Our assumption is that this function will mainly be used in situations where `endWeeklySettle` will not work.

3.4. The `costSharePrice` is not properly maintained

Target	VaultV2, WooSuperChargerVaultV2		
Category	Business Logic	**Severity**	Low
Likelihood	N/A	**Impact**	Low

Description

The `costSharePrice` is used in the VaultV2 and WooSuperChargerVaultV2 contracts to calculate the user's share price when depositing funds. The `costSharePrice` is calculated as the weighted average share price when the user deposits the funds.

It is calculated upon depositing funds, where the user is issued new shares. It is, however, not updated when the user transfers their shares to another user. This means that the `costSharePrice` is not properly maintained when the user transfers their shares to another user.

If User A has never called deposit, having their `costSharePrice = 0` while User B has called deposit (so User B's `costSharePrice > 0`) and then manually transfers their shares to User A, then User A will have the `balance > 0` but their `costPerSharePrice` will still be zero.

Impact

The impact is strictly related to user experience at the moment, as the `costSharePrice` is used for displaying purposes only. For posterity, however, we note that proper accounting is a good practice to follow, as it can prevent future issues.

Recommendations

We recommend properly accounting for the transfer case, where the `costSharePrice` is not properly maintained.

Remediation

This issue has been acknowledged by WOOFI, who noted the following:

> CostSharePrice couldn't handle the transfer case; We knew it and `costPerSharePrice` is used for displaying purposes. Vault v1 doesn't have this field, only after a few power users want to see their PnL for each vault, then we considered adding this field in V2.

This issue has been acknowledged by WOOFI.

WOOFI states:

> Currently known issue; it's only for displaying purposes, and we're noticing users in website that transferring the WE token may cause the PNL miscalculated.

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. Instant withdraw cap can be bypassed

The `instantWithdrawCap = instantWithdrawCap + amount / 10;` from WooSuperChargerVaultV2 can be bypassed by supplying an amount that is larger than 10, as it would underflow the `instantWithdrawCap` variable. We recommend setting a minimal deposit amount to prevent this from happening.

4.2. Similarities to ERC-4626 first-deposit issue

The Vault contracts(i.e VaultV2, WooSuperchargerVaultV2) are particularly similar to the ERC-4626 vaults. The similarities include the potential issue of first-deposit, where a malicious user can front-run the first deposit of a legitimate user, and inflate their worth of shares to withdraw more than they should, leaving the legitimate user with nothing to withdraw.

To mitigate against this issue, we recommend the following:

- Create "dead shares" upon the first liquidity deposit. This can be done in multiple ways, one of them being that the first deposit has to be performed by a trusted party and the shares are sent to a dead address.
- Keep track of assets held by the vault internally, rather than rely on the `balanceOf` function. This way, the donated assets cannot influence the vault's internal accounting.

4.3. Slippage check not performed during `compoundReward` function

The `compoundRewards` function from `RewardCampaignManager` does not perform slippage checks when swapping the reward token for WOO tokens. Slippage checks are an important part of ensuring that the swap occurs on favorable terms. Without slippage checks, the contract is exposed to wild price fluctuations that could result in a significant loss of funds, depending on the traded amounts.

```
function compoundRewards(address _user) public onlyAdmin {
    // ...
    // ...
    wooAmount += wooPP.swap(
        _rewarder.rewardToken(), // fromToken
        woo, // toToken
        rewardAmount, // fromAmount
        0, // minToAmount
        // @audit-issue no slippage checks.
        selfAddr, // to
        selfAddr // rebateTo
    );
    // ...
}
```

We recommend adding slippage checks to the `compoundRewards` function to ensure that the swap occurs on favorable terms.

5. Threat Model

This provides a full threat model description for various functions. As time permitted, we analyzed each function in the 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: BaseStrategy.sol

Function: `inCaseTokensGetStuck(address stuckToken)`

This allows the owner to recover any token sent to the contract by mistake.

Inputs

- `stuckToken`
  - **Control**: Fully controlled by the owner.
  - **Constraints**: Checked to be either native or some ERC-20 token.
  - **Impact**: The token to be recovered.

Branches and code coverage

Intended branches

- Assume malicious intent on the behalf of the caller.
  - ☐ Test coverage

Negative behavior

- Should not be callable by anyone other than the owner.
  - ☑ Negative test

5.2. Module: StrategyAave.sol

Function: `deposit()`

This allows depositing into the Aave strategy.

Branches and code coverage

Intended branches

- Assume that funds have already been transferred in the strategy.

  - ☑ Test coverage
- Deposit the funds in the Aave Pool on behalf of the strategy.
  - ☑ Test coverage

Negative behavior

- Should not allow anyone other than the vault to call this function. Currently not enforced.
  - □ Negative test

Function: `emergencyExit()`

This allows the admin to perform an emergency exit of the vault. This function will withdraw all funds from the strategy and transfer them to the vault.

Branches and code coverage

Intended branches

- Assume no malicious intent on the behalf of the admin.
  - ☑ Test coverage

Negative behavior

- Should not be callable by anyone other than the admin or pauser role.
  - ☑ Negative test

Function: `harvest()`

This allows harvesting rewards from the Aave Pool.

Branches and code coverage

Intended branches

- Harvest all rewards to the vault.
  - ☑ Test coverage

Negative behavior

- Should not allow anyone other than the vault or EOA to call this function.
  - ☑ Negative test

Function: `retireStrat()`

This allows the vault to retire the strategy.

Branches and code coverage

Intended branches

- Ensure that no rewards are still left in the Aave Pool.
  - □ Test coverage
- Withdraw everything from the Aave Pool.
  - ☑ Test coverage
- Transfer all funds to the vault.
  - ☑ Test coverage

Negative behavior

- Should not be callable by anyone other than the vault.
  - ☑ Negative test

Function: `withdraw(uint256 amount)`

This allows the vault to withdraw funds from the strategy.

Inputs

- `amount`
  - **Control**: Fully controlled by the vault — assumed that prior checks are done beforehand.
  - **Constraints**: Check if the amount is greater than zero.
  - **Impact**: The amount to withdraw from the strategy.

Branches and code coverage

Intended branches

- Charge withdrawal fees if the amount to be withdrawn is greater than the fee. Currently, the fees are charged regardless of the amount to be withdrawn.
  - □ Test coverage
- If the amount to withdraw is greater than the balance of the strategy, withdraw the remaining amount from the Aave Pool.
  - ☑ Test coverage
- Transfer the remaining amount to the vault.
  - ☑ Test coverage

Negative behavior

- Caller is a service admin.
  - ☑ Negative test

5.3. Module: VaultV2.sol

Function: `deposit(uint256 amount)`

This allows a user to deposit funds into the vault.

Inputs

- `amount`
  - **Control**: Fully controlled by the caller.
  - **Constraints**: Checked that the user can afford the deposit.
  - **Impact**: The amount of funds to be deposited.

Branches and code coverage

Intended branches

- Check that `msg.value` matches `amount` in the case of a native deposit.
  - ☑ Test coverage
- Ensure that `msg.value` is zero in the case of an ERC-20 deposit.
  - ☑ Test coverage
- Ensure that the strategy is not paused.
  - ☑ Test coverage
- Deposit the native tokens as WETH.
  - ☑ Test coverage
- Transfer the ERC-20 tokens in the case of an ERC-20 deposit.
  - ☑ Test coverage
- Mint the adequate amount of shares to the user.
  - ☑ Test coverage
- Update the `costSharePrice` of the user.
  - ☑ Test coverage
- Call the `earn` function so that the vault can earn yield on the freshly deposited funds.
  - ☑ Test coverage

Negative behavior

- Should not allow the user to deposit more than they can afford.
  - ☑ Negative test

Function: `earn()`

This allows the vault to earn yield on the deposited funds.

Branches and code coverage

Intended branches

- Transfer the funds into the strategy.
  - ☑ Test coverage
- Call the `deposit` function of the strategy.
  - ☑ Test coverage

Negative behavior

- Should not be callable by anyone other than the admin. Currently not enforced, and this can lead to a constant leach of funds due to accrued fees. Currently not performed.
  - □ Negative test

Function: `inCaseNativeTokensGetStuck()`

This allows the owner to recover native tokens sent to the contract by mistake.

Branches and code coverage

Intended branches

- Assumes malicious intent on the behalf of the caller.
  - □ Test coverage

Negative behavior

- Should not be callable by anyone other than the owner.
  - ☑ Negative test

Function: `inCaseTokensGetStuck(address stuckToken)`

This allows the owner to recover any token sent to the contract by mistake.

Inputs

- `stuckToken`
  - **Control**: Fully controlled by the owner.
  - **Constraints**: Checked to be either native or some ERC-20 token.
  - **Impact**: The token to be recovered.

Branches and code coverage

Intended branches

- Assumes malicious intent on the behalf of the caller.
  - □ Test coverage

Negative behavior

- Should not be callable by anyone other than the owner.
  - ☑ Negative test

Function: `proposeStrat(address _implementation)`

This allows the admin to propose a strategy for upgrade.

Inputs

- `_implementation`
  - **Control**: Fully controlled by caller.
  - **Constraints**: Ensure that it matches the desired parameters (i.e., want token).
  - **Impact**: The new implementation.

Branches and code coverage

Intended branches

- Ensure that enough time has passed based on the `approvalDelay`.
  - □ Test coverage
- Ensure that the `_implementation`'s vault matches the current vault.
  - ☑ Test coverage
- Ensure that the `_implementation`'s want token matches the current want token.
  - ☑ Test coverage

Negative behavior

- Should not allow anyone other than the admin to call this function.
  - ☑ Negative test

Function: `upgradeStrat()`

This function facilitates the upgrade of a strategy.

Branches and code coverage

Intended branches

- Assumed that the current strategy has no more funds or rewards to claim. Currently not

explicitly checked.
  - □ Test coverage
- Check whether the current implementation is set.
  - ☑ Test coverage
- Check if the proposed time has passed.
  - ☑ Test coverage
- Retire the current strategy.
  - ☑ Test coverage
- Update the strategy to the new implementation.
  - ☑ Test coverage
- Deposit funds into the new strategy. Performed in `earn`.
  - ☑ Test coverage

Negative behavior

- Should not be callable by anyone other than the admin.
  - ☑ Negative test

Function: `withdraw(uint256 shares)`

This allows a user to withdraw their funds from the vault.

Inputs

- `shares`
  - **Control**: Fully controlled by the caller.
  - **Constraints**: Checked that the caller can afford.
  - **Impact**: The amount of shares to be burned by `msg.sender`.

Branches and code coverage

Intended branches

- Ensure that `msg.sender` can afford the withdrawal.
  - ☑ Test coverage
- Calculate the amount to be withdrawn.
  - ☑ Test coverage
- Burn the calculated amount of shares from the caller.
  - ☑ Test coverage
- If there are not enough funds for the user to withdraw, withdraw the funds from the strategy.
  - ☑ Test coverage
- Send funds back to the user. In the case of WETH, withdraw WETH and transfer ETH to the user. Otherwise, transfer want ERC-20 tokens to the user.

☑ Test coverage

Negative behavior

- Should not allow `msg.sender` to deposit more than they can afford. Enforced through the `balanceOf` check and the explicit `_burn`.
  ☑ Negative test

5.4. Module: WooLendingManager.sol

Function: `borrow(uint256 amount)`

This allows borrowing and depositing the borrowed amount into WooPP.

Inputs

- `amount`
  - **Control**: Fully controlled by the caller.
  - **Constraints**: Must be greater than 0.
  - **Impact**: The amount to be borrowed.

Branches and code coverage

Intended branches

- Assumed there is some sort of arrangement or collateralization against borrowing between the protocol and the borrower.
  ☐ Test coverage
- Accrue the interest before borrowing.
  ☑ Test coverage
- Increase the `borrowedPrincipal` by the amount borrowed.
  ☑ Test coverage
- Borrow the amount from the SuperChargerVault.
  ☑ Test coverage
- Ensure that the difference between the balance before and after are equal to the amount borrowed.
  ☑ Test coverage
- Approve and deposit the borrowed amount into WooPP.
  ☑ Test coverage

Negative behavior

- Should not allow anyone other than a borrower to call this.
  ☑ Negative test

Function: `inCaseTokenGotStuck(address stuckToken)`

This allows the owner to recover any token sent to the contract by mistake.

Inputs

- `stuckToken`
  - **Control**: Fully controlled by the owner.
  - **Constraints**: Checked to be either native or some ERC-20 token.
  - **Impact**: The token to be recovered.

Branches and code coverage

Intended branches

- Assume malicious intent on the behalf of the caller.
  - ☐ Test coverage

Negative behavior

- Should not be callable by anyone other than the owner.
  - ☑ Negative test

Function: `repayPrincipal(uint256 _principal)`

This allows the repayment of the principal amount by custom amount.

Inputs

- `_principal`
  - **Control**: Full control by the caller.
  - **Constraints**: None.
  - **Impact**: The amount of principal to be repaid by the caller.

Branches and code coverage

Intended branches

- Accrue the interest before repaying.
  - ☑ Test coverage
- Repay by transferring in the funds from the caller.
  - ☑ Test coverage
- Update the `borrowedInterest` and `borrowedPrincipal` state variables by subtracting

the repaid amount.

☑ Test coverage

Negative behavior

- Should not allow anyone other than the borrower to repay.
  - ☑ Negative test
- Should not allow paying more than what is borrowed. Ensured through the `borrowedPrincipal` variable.
  - ☑ Negative test

5.5. Module: WooStakingCompounder.sol

Function: `compound(uint256 start, uint256 end)`

This is a utility function for compounding a range of users. Replicates the range logic from `allUsers(start, end)`. This is an admin-only function.

Inputs

- `start`
  - **Control**: Fully controllable by the admin.
  - **Constraints**: Should be less than `end` and strictly less than the length of the `users` set.
  - **Impact**: The index of the first address to include in the compound.
- `end`
  - **Control**: Fully controllable by the admin.
  - **Constraints**: Should be larger than `start` and up to the length of the `users` set.
  - **Impact**: The index to stop at (and not include) when compounding.

Branches and code coverage

Intended branches

- Called by admin for a single user.
  - ☑ Test coverage
- Called by admin with >1 users.
  - ☑ Test coverage

Negative behavior

- Called by non-admin.
  - □ Negative test
- Called with invalid indexes or `start>end`.
  - □ Negative test

5.6. Module: WooStakingController.sol

Function: `_nonblockingLzReceive(uint16, bytes, uint64, bytes _payload)`

This receives LZ events from the proxy and handles recognized events like staking, unstaking, compounding, and toggling the autocompound feature.

Inputs

The sender is authenticated earlier before arriving at this function, which means that inputs are not directly controllable. Amounts are controllable where those are used, but they follow restrictions set in the proxy.

Branches and code coverage

The function is not directly callable from test code, but it acts as a thin wrapper around the functions found in WooStakingManager.sol and provides authorized calls for them through this interface.

5.7. Module: WooStakingLocal.sol

Function: `compoundAll()`

This is a thin wrapper around the `WooStakingManager->compoundAll()` function.

Function: `compoundMP()`

This is a thin wrapper around the `WooStakingManager->compoundMP()` function.

Function: `emergencyUnstake()`

This allows the caller to unstake its tokens in an emergency. Does not unstake Woo in the staking manager. Emergencies can only be enabled by the admin.

Branches and code coverage

Intended branches

- Called during an emergency.
  - □ Test coverage

Negative behavior

- Called outside an emergency.

- □ Negative test

Function: `inCaseTokenGotStuck(address stuckToken)`

This allows the owner to recover any token sent to the contract by mistake.

Inputs

- `stuckToken`
  - **Control**: Fully controlled by the owner.
  - **Constraints**: Checked to be either native or some ERC-20 token.
  - **Impact**: The token to be recovered.

Branches and code coverage

Intended branches

- Called by the owner to retrieve stuck native tokens.
  - □ Test coverage
- Called by the owner to retrieve stuck ERC-20 tokens.
  - □ Test coverage

Negative behavior

- Should not be callable by anyone other than the owner.
  - ☑ Negative test

Function: `setAutoCompound(bool _flag)`

This enables or disables the automatic compound feature for the sender.

Inputs

- `_flag`
  - **Control**: Fully controlled by the caller.
  - **Constraints**: None.
  - **Impact**: Decides if autocompound should be enabled or disabled.

Branches and code coverage

Intended branches

- Enable autocompound.
  - □ Test coverage
- Disable autocompound.
  - □ Test coverage

Function call analysis

- `this.stakingManager.setAutoCompound(_user, _flag)`
  - **What is controllable?** The flag is controllable.
  - **If the return value is controllable, how is it used and how can it go wrong?** There is no return value, but it should have one.
  - **What happens if it reverts, reenters or does other unusual control flow?** Disabling compounding will call `compounder.removeUser(_user)` inside the WooStakingCompounder module. In turn, `removeUser` will call `_removeUser`, which returns a boolean stating if the user was actually removed or not. Removal can fail if the user is not in the list of users with autocompound or if the user is still under cooldown (in which a `RemoveAbortedInCooldown` event will be emitted). However, this return code is never checked, and the final `SetAutoCompoundOnLocal` event will still be emitted despite the removal failing.

Function: `stake(uint256 _amount)`

This stakes an amount on behalf of the caller. The stake is added to the current `balances`.

Inputs

- `_amount`
  - **Control**: Fully controlled by the caller.
  - **Constraints**: Cannot exceed the amount of tokens approved for the contract.
  - **Impact**: The amount to stake.

Branches and code coverage

Intended branches

- Stake the exact approved amount.
  - □ Test coverage
- Stake less than the approved amount.
  - □ Test coverage

Negative behavior

- Stake more than the approved amount.
  - □ Negative test

Function: `unstakeAll()`

This is a utility function to unstake the entire balance for the caller.

Function: `unstake(uint256 _amount)`

This unstakes some amount on behalf of the sender.

Inputs

- `_amount`
    - **Control**: Fully controlled by the caller.
    - **Constraints**: Cannot exceed the caller's staked balance.
    - **Impact**: The amount to unstake.

5.8. Module: WooStakingManager.sol

Function: `claimRewards(address _user)`

This manually claims rewards on behalf of a user address. It bypasses restrictions if autocompounding is enabled. Admin only.

Inputs

- `_user`
    - **Control**: Fully controlled by the caller.
    - **Constraints**: None.
    - **Impact**: The address that will get its rewards claimed.

Branches and code coverage

Intended branches

- Called by admin.
    - ☑ Test coverage

Negative behavior

- Called by non-admin.
    - ☑ Negative test

Function: `claimRewards()`

This manually claims rewards for the sender. Cannot be used if autocompounding is enabled.

Branches and code coverage

Intended branches

- Called by staker with autocompounding disabled.
  - ☑ Test coverage

Negative behavior

- Called by staker with autocompounding enabled.
  - ☑ Negative test

Function: `compoundAllForUsers(address[] _users)`

This is the same as `compoundAll(address _user)`, except for a list of users. It reimplements `compoundRewards(_user)` in a more optimized way for cases where there are multiple users, by only doing one swap per rewarder. It is an admin-only function.

Inputs

- `_users`
  - **Control**: Fully controllable by the admin.
  - **Constraints**: None.
  - **Impact**: The list of users to run compound for.

Branches and code coverage

Intended branches

- Call with multiple users.
  - ☑ Test coverage

Negative behavior

- Call with zero users.
  - □ Negative test
- Direct call by non-admin.
  - □ Negative test

Function: `compoundAll(address _user)`

This is a utility function to compound all the MP and rewards for a given user.

Inputs

- `_user`
    - **Control**: Not controlled by a non-admin caller.
    - **Constraints**: It is set to the initial caller.
    - **Impact**: The address to compound all for.

Function: `compoundMP(address _user)`

This compounds a user's MP by updating rewards and debt, claiming and then restaking back to this contract.

Inputs

- `_user`
    - **Control**: Only controllable by the admin; otherwise, it is the initial caller.
    - **Constraints**: Set to initial caller.
    - **Impact**: The user to compound MP for.

Branches and code coverage

Intended branches

- Called by admin.
    ☑ Test coverage

Negative behavior

- Called by non-admin.
    ☑ Negative test

Function: `compoundRewards(address _user)`

This compounds all the rewards from every rewarder, claiming and swapping the rewards for WOO, then restaking them back.

Inputs

- `_user`

- **Control**: Not controllable by normal user. Called indirectly.
- **Constraints**: Set to the original caller's address for non-admins.
- **Impact**: The address to compound rewards for.

Branches and code coverage

Intended branches

- Called by admin for a user.
  ☑ Test coverage

Negative behavior

- Called by non-admin.
  ☑ Negative test

Function: `setAutoCompound(address _user, bool _flag)`

This enables or disables automatic compounding for a given user. The user is only added if its current balance meets the threshold criteria set in the compounder module.

Inputs

- `_user`
  - **Control**: Fully controlled by an admin but limited to be the initial caller otherwise.
  - **Constraints**: None, but has no direct effect if the user has no balance or does not exist. The event is always emitted.
  - **Impact**: The address to attempt enabling or disabling compounding for.
- `_flag`
  - **Control**: Fully controlled by the caller.
  - **Constraints**: None.
  - **Impact**: Decision to enable or disable automatic compounding.

Branches and code coverage

Intended branches

- Enable automatic compounding.
  □ Test coverage
- Disable automatic compounding.
  □ Test coverage

Negative behavior

- Caller is not a service admin.
  - □ Negative test

Function call analysis

- `this.compounder.addUserIfThresholdMeet(_user)`
  - **What is controllable?** Fully controlled by an admin but limited to be the initial caller otherwise.
  - **If the return value is controllable, how is it used and how can it go wrong?** Return value tells if the user was added or not, but it is ignored. The event will be produced even if the user was not added.
  - **What happens if it reverts, reenters or does other unusual control flow?** Cannot revert, but attempting to add a user will always produce the event.
- `this.compounder.removeUser(_user)`
  - **What is controllable?** Fully controlled by an admin but limited to be the initial caller otherwise.
  - **If the return value is controllable, how is it used and how can it go wrong?** Same as for `addUserIfThresholdMeet`, but removal can also fail if the user is under cooldown. The return value is ignored, and all events are produced as normal.
  - **What happens if it reverts, reenters or does other unusual control flow?** Same as above.

Function: `stakeWoo(address _user, uint256 _amount)`

This stakes WOO tokens for the given user. Only indirectly callable from, for example, WooStakingController, which is an admin for this contract.

Inputs

- `_user`
  - **Control**: Fully controlled by the caller, through the function that stakes on behalf of another address.
  - **Constraints**: None.
  - **Impact**: The address credited with the stake.
- `_amount`
  - **Control**: Fully controlled by the caller.
  - **Constraints**: Nonzero.
  - **Impact**: The amount to stake.

Branches and code coverage

Intended branches

- Caller stakes a nonzero amount.
  - ☑ Test coverage

Negative behavior

- Caller is not the admin.
  - ☑ Negative test
- Caller stakes zero tokens.
  - □ Negative test

Function: `unstakeWoo(address _user, uint256 _amount)`

This unstakes WOO tokens for the given user. Only indirectly callable from, for example, WooStakingController, which is an admin for this contract. Also, it removes a proportional amount of MP tokens.

Inputs

- `_user`
  - **Control**: Not controllable by the initial caller.
  - **Constraints**: Limited to be the initial caller address.
  - **Impact**: The address debited with the unstake.
- `_amount`
  - **Control**: Fully controlled by the caller.
  - **Constraints**: Nonzero.
  - **Impact**: The amount to unstake.

Branches and code coverage

Intended branches

- Single unstake.
  - ☑ Test coverage
- Repeated unstake.
  - ☑ Test coverage

Negative behavior

- Unstake a larger amount than the user has available.
  - ☑ Negative test

Function call analysis

- `this.compounder.removeUserIfThresholdFail(_user)`
  - **What is controllable?** Not controllable.
  - **If the return value is controllable, how is it used and how can it go wrong?** Return value is ignored.
  - **What happens if it reverts, reenters or does other unusual control flow?** Cannot revert. User is removed from the compounder's `users` list if the new staked balance falls under `autoCompThreshold`.
- `EnumerableSet.remove(this.stakers, _user)`
  - **What is controllable?** Not controllable.
  - **If the return value is controllable, how is it used and how can it go wrong?** Return value is ignored.
  - **What happens if it reverts, reenters or does other unusual control flow?** Cannot revert. If the user does not exist in the list, a boolean is returned (but ignored). Normally this cannot happen, because a user has to stake in order to unstake, and you can neither stake nor unstake an amount of zero.

5.9. Module: WooStakingProxy.sol

Function: `compoundAll()`

This sends an LZ message of type `ACTION_COMPOUND_ALL` on behalf of the sender.

Function: `compoundMP()`

This sends an LZ message of type `ACTION_COMPOUND_MP` on behalf of the sender.

Function: `emergencyUnstake()`

This unstakes the entire balance of the sender. Can only be used in emergencies. Does not produce LZ events.

Branches and code coverage

Intended branches

- Called during an emergency.
  - □ Test coverage

Negative behavior

- Called when there is no emergency.
  - □ Negative test

Function: `inCaseTokenGotStuck(address stuckToken)`

This allows the owner to recover any token sent to the contract by mistake.

Inputs

- `stuckToken`
    - **Control**: Fully controlled by the owner.
    - **Constraints**: Checked to be either native or some ERC-20 token.
    - **Impact**: The token to be recovered.

Branches and code coverage

Intended branches

- Called by owner to retrieve stuck native tokens.
    - □ Test coverage
- Called by owner to retrieve stuck ERC-20 tokens.
    - □ Test coverage

Negative behavior

- Should not be callable by anyone other than the owner.
    - ☑ Negative test

Function: `setAutoCompound(bool _flag)`

This enables or disables the automatic compound feature for the sender. Produces an LZ event with type `ACTION_SET_AUTO_COMPOUND`.

Inputs

- `_flag`
    - **Control**: Fully controlled by the caller.
    - **Constraints**: None.
    - **Impact**: Decides if autocompound should be enabled or disabled.

Function: `stake(address _user, uint256 _amount)`

This is the same as `stake(uint256 _amount)`, but it stakes `msg.sender` want token on behalf of `_user`.

Inputs

- `_user`
  - **Control**: Fully controlled by the caller.
  - **Constraints**: None. Can even stake on behalf of `address(0)`.
  - **Impact**: The address that is credited with the balance for the staking.
- `_amount`
  - **Control**: Fully controlled by the caller.
  - **Constraints**: Has to be more than or equal to the amount sent in the transaction.
  - **Impact**: The amount to stake, which is added to the current balance of `_user`.

Branches and code coverage

Intended branches

- Stake an amount equal to the funds sent.
  - □ Test coverage

Negative behavior

- Stake more than what is being sent.
  - □ Negative test

Function: `stake(uint256 _amount)`

This is a proxy function for staking a given amount. Sends an LZ message with type `ACTION_STAKE`.

Inputs

- `_amount`
  - **Control**: Fully controlled by the caller.
  - **Constraints**: Has to be more than or equal to the amount sent in the transaction.
  - **Impact**: The amount to stake, which is added to the current balance of the sender.

Branches and code coverage

Intended branches

- Stake an amount equal to the funds sent.
  - □ Test coverage

Negative behavior

- Stake more than what is being sent.
  - □ Negative test

Function: `unstakeAll()`

This is a utility function that unstakes the entire balance of the sender.

Function: `unstake(uint256 _amount)`

This unstakes some amount from the sender's balance and sends funds to the sender. Produces an LZ event of type `ACTION_UNSTAKE`.

Inputs

- `_amount`
  - **Control**: Fully controlled by the caller.
  - **Constraints**: Cannot exceed the sender's balance.
  - **Impact**: The amount to unstake.

Branches and code coverage

Intended branches

- Called with the exact balance amount.
  - □ Test coverage
- Called with less than the balance amount.
  - □ Test coverage

Negative behavior

- Called with more than the balance amount.
  - □ Negative test

5.10. Module: WooSuperChargerVaultV2.sol

Function: `deposit(uint256 amount)`

This allows depositing in the SuperChargerVault.

Inputs

- `amount`
    - **Control**: Fully controlled.
    - **Constraints**: User needs to afford `amount` tokens.
    - **Impact**: The amount of tokens to deposit.

Branches and code coverage

Intended branches

- Ensure that the deposit amount is greater than 10 so that the `instantWithdrawCap` is updated properly.
    - □ Test coverage
- Accrue interest in the lending manager.
    - ☑ Test coverage
- Calculate the amount of shares to mint.
    - ☑ Test coverage
- Update the cost share price of the receiver.
    - ☑ Test coverage
- If `want == weth`, deposit the amount as native tokens in the reserve vault.
    - ☑ Test coverage
- If `want != weth`, transfer the amount as ERC-20 tokens to the reserve vault.
    - ☑ Test coverage
- Mint the adequate amount of shares to the receiver.
    - ☑ Test coverage
- Update the instant withdraw cap.
    - ☑ Test coverage

Negative behavior

- Should not allow depositing an amount that is not afforded by the user.
    - ☑ Negative test

Function: `endWeeklySettle()`

This performs the weekly settlement for the vault.

Branches and code coverage

Intended branches

- Should not be able to DOS by adding small requests. This should be enforced via `maxWithdrawCount` from the `requestWithdraw` function.
    - ☑ Test coverage

- Ensure that it is in the settling phase.
    - ☑ Test coverage
- Ensure that the weekly repay is cleared.
    - ☑ Test coverage
- Set the settling flag to false.
    - ☑ Test coverage
- If there is a total withdraw amount, withdraw it from the reserve vault.
    - ☑ Test coverage
- If the want is WETH, deposit the total withdraw amount.
    - ☑ Test coverage
- Ensure that the available balance is enough for the total withdraw amount.
    - ☑ Test coverage
- Iterate over all the users and add the withdraw amount to the withdraw manager.
    - ☑ Test coverage
- Reset the requested withdraw shares and deplete the users' requests array.
    - ☑ Test coverage
- Remove the users from the request users set.
    - ☑ Test coverage
- Burn the total collected shares.
    - ☑ Test coverage
- Reset the total requested shares.
    - ☑ Test coverage
- Transfer the total withdraw amount to the withdraw manager.
    - ☑ Test coverage
- Accrue the interest.
    - ☑ Test coverage
- Update the `instantWithdrawCap` to 10% of the total balance.
    - ☑ Test coverage

Negative behavior

- Should not be callable by anyone other than the admin.
    - ☑ Negative test

Function: `inCaseTokenGotStuck(address stuckToken)`

This allows owner to recover any token sent to the contract by mistake.

Inputs

- `stuckToken`
    - **Control**: Fully controlled by the owner.
    - **Constraints**: Checked to be either native or some ERC-20 token.

- **Impact**: The token to be recovered.

Branches and code coverage

Intended branches

- Assumes malicious intent on the behalf of the caller.
  - □ Test coverage

Negative behavior

- Should not be callable by anyone other than the owner.
  - ☑ Negative test

Function: `instantWithdraw(uint256 amount)`

This allows a user to instantly withdraw from the SuperChargerVault.

Inputs

- `amount`
  - **Control**: Fully controlled.
  - **Constraints**: User needs to afford `amount` tokens in terms of the shares that are to be burned.
  - **Impact**: The amount of tokens to withdraw.

Branches and code coverage

Intended branches

- Accrue interest in the lending manager.
  - ☑ Test coverage
- Calculate the amount of assets to withdraw.
  - ☑ Test coverage
- Check if the instant withdraw cap is reached.
  - ☑ Test coverage
- If the caller is not the owner of the funds, spend the allowance of the owner.
  - ☑ Test coverage
- Burn the shares of the owner.
  - ☑ Test coverage
- Withdraw the reserve shares.
  - ☑ Test coverage
- Calculate the fees and deduct them from the withdrawn amount — also, transfer the fees to the treasury.

☑ Test coverage

- Transfer the remaining amount to the owner.
  ☑ Test coverage
- Update the instant withdraw amount by adding the withdrawn amount.
  ☑ Test coverage

Negative behavior

- Should not allow withdrawing more than the cap.
  ☑ Negative test
- Should not allow withdrawing if the user cannot afford the shares. Enforced through `_burn`.
  ☑ Negative test

Function: `migrateReserveVault(address _vault)`

This allows the owner to perform a migration of the vault.

Inputs

- `_vault`
  - **Control**: Fully controlled by owner.
  - **Constraints**: Checked that it is not the zero address.
  - **Impact**: The new vault address.

Branches and code coverage

Intended branches

- Withdraw everything from the current vault.
  ☑ Test coverage
- Deposit everything to the new vault.
  ☑ Test coverage
- Update the `reserveVault` address.
  ☑ Test coverage
- Deposit the reserve amount to the new vault. Native in the case that the want is WETH, otherwise ERC-20.
  ☑ Test coverage

Negative behavior

- Should not be callable by anyone other than the owner.
  ☑ Negative test

Function: `migrateToNewVault()`

This allows a user to migrate to a new vault.

Branches and code coverage

Intended branches

- Mint new shares to the receiver. Handled in the `deposit` call with the new user as a parameter.
  - ☑ Test coverage
- Burn the shares of the current vault for the caller.
  - ☑ Test coverage
- Deposit the amount of tokens in the new vault.
  - ☑ Test coverage

Negative behavior

- Should not allow user migrating if there is no migration vault set.
  - ☑ Negative test

Function: `requestWithdraw(uint256 amount)`

This allows a user to perform a withdraw request.

Inputs

- `amount`
  - **Control**: Fully controlled by the user.
  - **Constraints**: User needs to afford the amount of shares to be withdrawn.
  - **Impact**: The amount of shares to be withdrawn.

Branches and code coverage

Intended branches

- Ensure that it is not settling phase.
  - ☑ Test coverage
- Ensure that the length of the user requests is less than the maximum withdraw count.
  - ☑ Test coverage
- Transfer the shares from the user to the contract.
  - ☑ Test coverage
- Add the shares to the requested withdraw shares mapping.
  - ☑ Test coverage

- Increase the total requested shares.
  - ☑ Test coverage
- Add the user to the request users set.
  - ☑ Test coverage

Negative behavior

- Should not allow withdrawing if the amount is zero.
  - ☑ Negative test
- Should not allow withdrawing if the user cannot afford the shares.
  - ☑ Negative test

5.11. Module: WooWithdrawManagerV2.sol

Function: `addWithdrawAmount(address user, uint256 amount)`

This allows the SuperChargerVault to add a withdrawal amount for a user.

Inputs

- `user`
  - **Control**: Fully controlled by the SuperChargerVault.
  - **Constraints**: None.
  - **Impact**: The user whose withdrawal amount is to be updated.
- `amount`
  - **Control**: Fully controlled by the SuperChargerVault.
  - **Constraints**: None.
  - **Impact**: The amount to be added to the user's withdrawal amount.

Branches and code coverage

Intended branches

- Increase the `withdrawAmount` mapping for the user.
  - ☑ Test coverage
- Assumed that prior checks are performed before calling this function.
  - ☑ Test coverage

Negative behavior

- Should not allow anyone other than the SuperChargerVault to call this function.
  - ☑ Negative test

Function: `inCaseTokenGotStuck(address stuckToken)`

This allows the owner to recover any token sent to the contract by mistake.

Inputs

- `stuckToken`
    - **Control**: Fully controlled by the owner.
    - **Constraints**: Checked to be either native or some ERC-20 token.
    - **Impact**: The token to be recovered.

Branches and code coverage

Intended branches

- Assumes malicious intent on the behalf of the caller.
    - □ Test coverage

Negative behavior

- Should not be callable by anyone other than the owner.
    - ☑ Negative test

Function: `withdraw()`

This allows withdrawing from the contract.

Branches and code coverage

Intended branches

- Ensure that a user can afford the withdrawal. Ensured through `withdrawAmount[msg.sender]`.
    - ☑ Test coverage
- Update the `withdrawAmount` mapping for the user.
    - ☑ Test coverage
- If the user has nothing to withdraw, return early.
    - ☑ Test coverage
- If `want == weth`, withdraw WETH and transfer ETH to the user.
    - ☑ Test coverage
- If `want != weth`, transfer want tokens to the user.
    - ☑ Test coverage

Negative behavior

- Should not allow transferring more than the user can afford.
  - ☑ Negative test

6. Assessment Results

At the time of our assessment, the reviewed code was already deployed to the Arbitrum mainnet.

During our assessment on the scoped WOOFI Stake contracts, we discovered four findings. One critical issue was found. Two were of medium impact and one was of low impact.

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 version reviewed during 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. These recommendations are not exhaustive, and we encourage our partners to consider them as a starting point for further discussion. We are happy to provide additional guidance and advice as needed.

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.