Zellic

Staking

Smart Contract Security Assessment

September 21, 2023

Prepared for:

Simon Mall

GammaSwap

Prepared by:

Ayaz Mammadov and Vlad Toie

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 GammaSwap from September 11th to September 19th, 2023. During this engagement, Zellic reviewed Staking'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:

- Can the vesting functionality be gamed for quick rewards?
- Can the reward tracker be fooled?
- Are there any staking incentives within the protocol that might have adverse effects?
- Is the deployment of the contracts safe and correct?

1.2 Non-goals and Limitations

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

- Ensuring that the reward distributor gets rewards
- The failure of underlying tokens that may be used in pairs
- Front-end components
- Infrastructure relating to the project
- Key custody

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

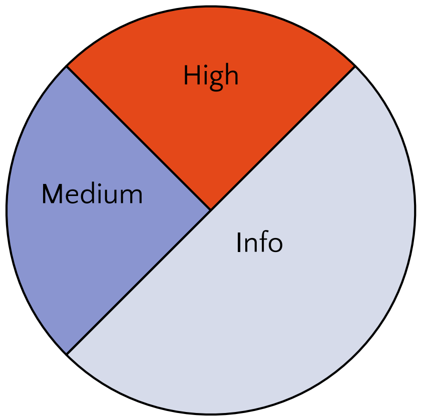
1.3 Results

During our assessment on the scoped Staking contracts, we discovered four findings. No critical issues were found. One finding was of high impact, one was of medium impact, and the remaining findings were informational in nature.

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

Breakdown of Finding Impacts

Impact Level	Count
Critical	0
High	1
Medium	1
Low	0
Informational	2



2 Introduction

2.1 About Staking

GammaSwap's Staking project refers to the reward/staking aspects of the GammaSwap protocol. This involves GS (native token), esGS (escrowed GS token), esGSb (escrowed GS token for borrowers), and the StakingRouter contract, which has all user-facing functions.

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, 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.

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:

Staking Contracts

Repository	https://github.com/gammaswap/v1-staking
Version	v1-staking: `26f89ed97673376de687eb33896c402dff979be4`
Programs	BonusDistributor.sol RewardDistributor.sol RewardTracker.sol StakingAdmin.sol StakingRouter.sol Vester.sol VesterNoReserve.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 two person-weeks. The assessment was conducted over the course of 7 calendar days.

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:

Ayaz Mammadov, Engineer
ayaz@zellic.io

Vlad Toie, Engineer
vlad@zellic.io

2.5 Project Timeline

The key dates of the engagement are detailed below.

September 11, 2023	Start of primary review period
September 19, 2023	End of primary review period

3 Detailed Findings

3.1 Vester incorrect burn

- **Target**: VesterNoReserve
- **Category**: Business Logic
- **Likelihood**: High
- **Severity**: High
- **Impact**: High

Description

Vesting is the process of locking tokens for a certain interval of time, after which the tokens are returned with rewards. The function `_updateVesting`, that is called to update vesting states burns `esToken`, which represent the users locked tokens, from the account. This is incorrect as locked esTokens are transferred to the Vesting contract when deposited.

```
function _updateVesting(address _account) private {
    uint256 amount = _getNextClaimableAmount(_account);
    lastVestingTimes[_account] = block.timestamp;

    if (amount == 0) {
        return;
    }

    // transfer claimableAmount from balances to cumulativeClaimAmounts
    _burn(_account, amount);
    cumulativeClaimAmounts[_account] = cumulativeClaimAmounts[_account]
    + amount;

    IRestrictedToken(esToken).burn(_account, amount);
}
```

Impact

If a user deposits more than half of their `esToken`, they cannot `claim` or `withdraw` more tokens without acquiring more esToken as it will revert due to the lack of tokens during the burn.

If the user has enough tokens to be burned (not deposited tokens), every time `_updat` `eVesting` is called, their esTokens will be burned, receiving no tokens in return.

Recommendations

Correct the logic to burn tokens from the Vester contract and not from the user.

Remediation

This issue has been acknowledged by GammaSwap, and a fix was implemented in commit a3672730.

3.2 Cancellation of `isDepositToken` still allows rewards to be claimed

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

Description

The `isDepositToken` mapping is used to validate whether a token is whitelisted to be staked in the contract.

```
function _stake(address _fundingAccount, address _account,
    address _depositToken, uint256 _amount) internal virtual {
    // ...
    require(isDepositToken[_depositToken], "RewardTracker: invalid
    _depositToken");

    IERC20(_depositToken).safeTransferFrom(_fundingAccount,
    address(this), _amount);

    // ...
}
```

A similar check is performed upon `unstaking` of tokens.

```
function _unstake(address _account, address _depositToken,
    uint256 _amount, address _receiver) internal virtual {
    // ...
    require(isDepositToken[_depositToken], "RewardTracker: invalid
    _depositToken");

    // ...

    _burn(_account, _amount);
    IERC20(_depositToken).safeTransfer(_receiver, _amount);
}
```

Thus, if the `isDepositToken` mapping is set to `False` after previously being `True`, any amount of tokens that have been staked in the contract will **not** be able to be unstaked.

Despite this, the rewards that have been accumulated will still be claimable.

Impact

The impact of this issue depends on the implementation of the rest of the protocol and several other considerations.

Since theoretically the `isDepositToken` can be called again to re-whitelist the token, the impact is diminished. However, in the case that the `isDepositToken` is a token that has been compromised and is no longer wanted by the system, this quick fix is no longer a viable alternative, and the issue becomes a severe problem, as the rewards are still accruing.

Recommendations

We recommend reconsidering the accrual of rewards for tokens that have been removed from the `isDepositToken` mapping. Essentially, they should not be considered towards the total amount of rewards that are claimable.

Remediation

This issue has been acknowledged by GammaSwap, and a fix was implemented in commit d29a27bb.

It is important to note, however, that the fix simply removes the `isDepositToken` check from the `_unstake` function. This could pose a security risk down the line if the `depositBalances` mapping is not properly updated on its own, as the `_depositToken` parameter is not checked for validity.

3.3 Check validity of parameters

- **Target**: StakingRouter
- **Category**: Business Logic
- **Likelihood**: Low
- **Severity**: Informational
- **Impact**: Informational

Description

Parameters such as the `_gsPool` in StakingRouter's functions could be checked for validity. For example,

```
function withdrawEsGsForPool(address _gsPool) external nonReentrant {
    IVester(poolTrackers[_gsPool].vester).withdrawForAccount(msg.sender);
}
```

lacks a check that the `_gsPool` is a valid address in the `poolTrackers` mapping.

Impact

Failure to properly check the validity of parameters could lead to unexpected behavior, which in this case would have resulted in a failed external call. It is a good security practice to ensure the validity of parameters before using them, especially when these refer to arbitrary addresses.

Recommendations

In the function above, the `_gsPool` parameter could be checked that it exists within the `poolTrackers` mapping. This would prevent the function from being called with an invalid `_gsPool` address.

```
function withdrawEsGsForPool(address _gsPool) external nonReentrant {
    require(poolTrackers[_gsPool].vester ≠ address(0), "StakingRouter:
        Pool not found");
    IVester(poolTrackers[_gsPool].vester).withdrawForAccount(msg.sender);
}
```

Remediation

This issue has been acknowledged by GammaSwap.

3.4 Set range limits for parameters

- **Target**: VesterNoReserve, Vester
- **Category**: Business Logic
- **Likelihood**: Low
- **Severity**: Informational
- **Impact**: Informational

Description

Important parameters, such as the cumulativeRewardDeductions and bonusRewards could benefit from assuring that the values they are set to are within a reasonable range. For example,

```
function setBonusRewards(address _account, uint256 _amount)
    external override nonReentrant {
    _validateHandler();
    bonusRewards[_account] = _amount;
}
```

does not have a check that the _amount is within a reasonable range. This could be remediated by adding a check that the _amount is less than or equal to a MAX_BONUS_REWARDS constant.

Where the MAX_BONUS_REWARDS constant could be defined as:

```
uint256 public constant MAX_CUMULATIVE_REWARD_DEDUCTIONS = 1000;
uint256 public constant MAX_BONUS_REWARDS = 1000;
```

Impact

Although this does not pose a direct security risk, as the functions can only be performed by a handler, it is a good security practice to ensure the validity of parameters before setting them. It also provides clarifies to the user that the contract they are interacting with has a limit on the values that can be set for particular parameters.

Recommendations

We recommend implementing range checks in both setCumulativeRewardDeductions and setBonusRewards.

Remediation

This issue has been acknowledged by GammaSwap.

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 Deployment failure could waste gas

The StakingAdmin contract contains the deployment of many contracts in certain functions such as `setupGsStaking`. In the case of a failure late in the deployment stage, a lot of funds in the form of gas could be lost.

One alternative could be to split up the deployment function, such that if one contract deployment fails then all other deployments are not reverted.

4.2 Testing coverage assessment

In our assessment of GammaSwap's test suite, we observed that while it provides adequate coverage for many aspects of the codebase, there are specific branches and code paths that appear to be under-tested or not covered at all.

Invariant Testing:

Invariant conditions are conditions that should always hold true, regardless of the state of the system. For example, the total supply of a token should always be equal to the sum of all token balances.

Invariant testing is a useful testing procedure to ensure that the system is functioning as expected. We recommend extending invariant tests for the `StakingRouter`, so that the invariant conditions of the system are always verified. Examples of invariant conditions here could be:

```
// fail if there's any difference between the ratio of what two users
    staked (assume they are random values) and the rewards they are
    entitled to

uint256 lpBalanceBeforeUser1 = GammaPoolERC20(pool).balanceOf(user1);
uint256 lpBalanceBeforeUser2 = GammaPoolERC20(pool).balanceOf(user2);

// deposit random amounts for user1 and user2
```

```
vm.prank(user1);
stakingRouter.stakeLp(address(pool), lpAmount1);
vm.prank(user2);
stakingRouter.stakeLp(address(pool), lpAmount2);

uint256 lpBalanceAfterUser1 = GammaPoolERC20(pool).balanceOf(user1);
uint256 lpBalanceAfterUser2 = GammaPoolERC20(pool).balanceOf(user2);

vm.warp(block.timestamp + 1 days);

uint256 claimableRewardsUser1
    = IRewardTracker(poolRewardTracker).claimable(user1);
uint256 claimableRewardsUser2
    = IRewardTracker(poolRewardTracker).claimable(user2);

// ensure that the ratio of rewards is the same as the ratio of deposits

assertEq(
    (lpBalanceBeforeUser1 - lpBalanceAfterUser1) / claimableRewardsUser1,
    (lpBalanceBeforeUser2 - lpBalanceAfterUser2) / claimableRewardsUser2
);

// ...
```

Integration Testing:

While unit tests are present and comprehensive, there is room for improvement in integration testing.

The interaction between various smart contract components and their combined behavior under various scenarios should be thoroughly examined. Integration tests can help identify unforeseen issues arising from the interplay of different modules.

We recommend specifically improving testing around the `StakingAdmin` to ensure that all setters and initializers are called in the correct order. This is especially important for the `StakingAdmin` as it is the "deployer" of most other contracts in the system.

In our assessment, we found that enhancing test coverage for these specific areas would further bolster the reliability and resilience of the project. We recommend expanding the test suite to include test cases addressing the above points.

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: BonusDistributor.sol

Function: `distribute()`

Distributes the rewards to the reward tracker contract.

Branches and code coverage (including function calls)

Intended branches

- Should increase the balance of the RewardTracker contract.
  - ☑ Test coverage
- Updates the last distribution time.
  - ☑ Test coverage
- Should only send the amount of rewards that are available. This means that if the contract has less rewards than the amount of rewards that should be distributed, it will send all the rewards that it has.
  - ☑ Test coverage

Negative behavior

- Should not be callable by anyone other than the RewardTracker contract.
  - ☑ Negative test
- Should not allow distribution if there are no rewards to distribute.
  - ☑ Negative test

Function: `setBonusMultiplier(uint256 _bonusMultiplierBasisPoints)`

Sets the bonus multiplier basis points.

Inputs

- `_bonusMultiplierBasisPoints`
  - **Control**: Fully controlled by the owner.
  - **Constraints**: N/A.
  - **Impact**: Sets the bonus multiplier basis points.

Branches and code coverage (including function calls)

Intended branches

- Sets the bonus multiplier basis points.
  ☑ Test coverage
- Calls `IRewardTracker.updateRewards()`.
  ☑ Test coverage
- Should be checked that it fits within some range. Currently not checked.
  ☐ Test coverage

Negative behavior

- Should not be callable by anyone other than the owner.
  ☑ Negative test
- Should revert if the last distribution time is invalid.
  ☑ Negative test

Function: `updateLastDistributionTime()`

Sets the last distribution time to the current block timestamp.

Branches and code coverage (including function calls)

Intended branches

- Sets the last distribution time to the current block timestamp.
  ☑ Test coverage

Negative behavior

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

5.2 Module: RewardDistributor.sol

Function: `distribute()`

Distributes the rewards to the RewardTracker contract.

Branches and code coverage (including function calls)

Intended branches

- Should increase the balance of the RewardTracker contract.
  - ☑ Test coverage
- Updates the last distribution time.
  - ☑ Test coverage
- Should only send the amount of rewards that are available. This means that if the contract has less rewards than the amount of rewards that should be distributed, it will send all the rewards that it has.
  - ☑ Test coverage

Negative behavior

- Should not be callable by anyone other than the RewardTracker contract.
  - ☑ Negative test
- Should not allow distribution if there are no rewards to distribute.
  - ☑ Negative test

Function: `updateLastDistributionTime()`

Sets the last distribution time to the current block timestamp.

Branches and code coverage (including function calls)

Intended branches

- Sets the last distribution time to the current block timestamp.
  - ☑ Test coverage

Negative behavior

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

5.3 Module: RewardTracker.sol

Function: `approve(address _spender, uint256 _amount)`

Allows approving allowances for transferring tokens.

Inputs

- `_spender`
  - **Control**: Fully controlled by the caller.
  - **Constraints**: None.
  - **Impact**: The account that will be allowed to transfer tokens.
- `_amount`
  - **Control**: Fully controlled by the caller.
  - **Constraints**: None.
  - **Impact**: The amount of tokens to be allowed to be transferred.

Branches and code coverage (including function calls)

Intended branches

- Sets the allowance of `_spender` to `_amount` for the `msg.sender`.
  ☑ Test coverage
- Check that `_spender` and `_owner` are not the zero address (in `_approve`).
  ☑ Test coverage

Negative behavior

- Should not allow approving tokens on behalf of others.
  ☑ Negative test

Function: `claimForAccount(address _account, address _receiver)`

Allows claiming of rewards on behalf of another account.

Inputs

- `_account`
  - **Control**: Fully controlled by the caller (handler).
  - **Constraints**: Only a handler can call this function.
  - **Impact**: The account that will be claimed for.
- `_receiver`
  - **Control**: Fully controlled by the caller (handler).
  - **Constraints**: None.
  - **Impact**: The destination for the claimed rewards.

Branches and code coverage (including function calls)

Intended branches

- Deletes the `claimableReward` for the `_account`.
  - ☑ Test coverage
- Transfers the `tokenAmount` of reward tokens to the `_receiver`.
  - ☑ Test coverage

Negative behaviour

- Should not be callable if `isPrivateClaimingMode` is `true`.
  - ☑ Negative test
- Should not leave any rewards to be claimed again.
  - ☑ Negative test

Function: `claim(address _receiver)`

Allows claiming of rewards.

Inputs

- `_receiver`
  - **Control**: Fully controlled by the caller.
  - **Constraints**: None.
  - **Impact**: The destination for the claimed rewards.

Branches and code coverage (including function calls)

Intended branches

- Deletes the `claimableReward` for `msg.sender`.
  - ☑ Test coverage
- Transfers the `tokenAmount` of reward tokens to the `_receiver`.
  - ☑ Test coverage

Negative behavior

- Should not be callable if `isPrivateClaimingMode` is `true`.
  - ☑ Negative test
- Should not leave any rewards to be claimed again.
  - ☑ Negative test

Function: `initialize(address[] _depositTokens, address _distributor)`

Initializes the contract.

Inputs

- `_depositTokens`
  - **Control**: Fully controlled by the owner.
  - **Constraints**: None.
  - **Impact**: Whitelists specified tokens for staking.
- `_distributor`
  - **Control**: Fully controlled by the owner.
  - **Constraints**: None.
  - **Impact**: Sets the RewardDistributor contract address.

Branches and code coverage (including function calls)

Intended branches

- Sets the `isInitialized` flag to `true`.
  - ☑ Test coverage
- Sets the `isDepositToken` flag to `true` for each `_depositToken`.
  - ☑ Test coverage
- Sets the `distributor` address to `_distributor`.
  - ☑ Test coverage

Negative behavior

- Should not be callable by anyone other than the owner.
  - ☑ Negative test
- Should not be callable if the contract is already initialized.
  - ☑ Negative test

Function: `setDepositToken(address _depositToken, bool _isDepositToken)`

Adds/removes a token from the deposit token whitelist.

Inputs

- `_depositToken`
  - **Control**: Fully controlled by the owner.
  - **Constraints**: None.
  - **Impact**: Whitelists/unwhitelists the specified token for staking.
- `_isDepositToken`
  - **Control**: Fully controlled by the owner.
  - **Constraints**: None.
  - **Impact**: Sets the `isDepositToken` flag to the specified value for the specified

token.

Branches and code coverage (including function calls)

Intended branches

- Sets the `isDepositToken` flag to the specified value for the specified token.
  ☑ Test coverage

Negative behavior

- Assumes there are no disruptive side effects. Currently, if `isDepositToken` is set to `false` for a token that has already been staked, the staked amount will be lost. However, the users can still claim their rewards.
  ☐ Negative test
- Should not be callable by anyone other than the owner.
  ☑ Negative test

Function: `stakeForAccount(address _fundingAccount, address _account, address _depositToken, uint256 _amount)`

Allows staking of tokens on behalf of another account.

Inputs

- `_fundingAccount`
  - **Control**: Fully controlled by the caller (handler).
  - **Constraints**: None.
  - **Impact**: The account that will be used to fund the staking.
- `_account`
  - **Control**: Fully controlled by the caller (handler).
  - **Constraints**: None.
  - **Impact**: The account that will be staked for.
- `_depositToken`
  - **Control**: Controlled by the caller (handler).
  - **Constraints**: Checked that the token is whitelisted (`isDepositToken`).
  - **Impact**: The token to be staked.
- `_amount`
  - **Control**: Controlled by the caller (handler).
  - **Constraints**: Checked that the amount is greater than zero. Also, in the `safeTransferFrom` call, it is checked that the `_fundingAccount` (in this case, `msg.sender`) has enough balance.

    - **Impact**: The amount of tokens to be staked.

Branches and code coverage (including function calls)

Intended branches

- Assumes no malicious intent on behalf of the handler.
  - ☑ Test coverage
- Checks that the token is whitelisted (`isDepositToken`).
  - ☑ Test coverage
- Checks that the amount is greater than zero.
  - ☑ Test coverage
- Checks that the `_fundingAccount` has enough balance (`safeTransferFrom`).
  - ☑ Test coverage
- Updates the rewards for the `_account`.
  - ☑ Test coverage
- Updates the `stakedAmounts` for the `_account`.
  - ☑ Test coverage
- Updates the `depositBalances` for the `_account`.
  - ☑ Test coverage
- Updates the `totalDepositSupply` for the `_depositToken`.
  - ☑ Test coverage
- Mints the `_amount` of tokens to the `_account`.
  - ☑ Test coverage

Negative behavior

- Should not be callable by anyone other than a handler.
  - ☑ Negative test
- Should not be callable if `isPrivateStakingMode` is `true`.
  - ☑ Negative test

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

Allows staking of tokens.

Inputs

- `_depositToken`
  - **Control**: Controlled by the caller.
  - **Constraints**: Checked that the token is whitelisted (`isDepositToken`).
  - **Impact**: The token to be staked.

- `_amount`
  - **Control**: Controlled by the caller.
  - **Constraints**: Checked that the amount is greater than zero. Also, in the `safeTransferFrom` call, it is checked that the `_fundingAccount` (in this case, `msg.sender`) has enough balance.
  - **Impact**: The amount of tokens to be staked.

Branches and code coverage (including function calls)

Intended branches

- Checks that the token is whitelisted (`isDepositToken`).
  ☑ Test coverage
- Checks that the amount is greater than zero.
  ☑ Test coverage
- Checks that the `_fundingAccount` (in this case, `msg.sender`) has enough balance (`safeTransferFrom`).
  ☑ Test coverage
- Updates the rewards for the `_account`.
  ☑ Test coverage
- Updates the `stakedAmounts` for the `_account`.
  ☑ Test coverage
- Updates the `depositBalances` for the `_account`.
  ☑ Test coverage
- Updates the `totalDepositSupply` for the `_depositToken`.
  ☑ Test coverage
- Mints the `_amount` of tokens to the `_account`.
  ☑ Test coverage

Negative behavior

- Should not be callable if `isPrivateStakingMode` is `true`.
  ☑ Negative test

Function: `transferFrom(address _sender, address _recipient, uint256 _amount)`

Allows transferring tokens from an account to another.

Inputs

- `_sender`
  - **Control**: Fully controlled by the caller.

    - **Constraints**: None — only checked that msg.sender has enough allowance for the particular _sender.
    - **Impact**: The account that will send the tokens.
- _recipient
    - **Control**: Fully controlled by the caller.
    - **Constraints**: None.
    - **Impact**: The account that will receive the tokens.
- _amount
    - **Control**: Fully controlled by the caller.
    - **Constraints**: Ensured that msg.sender has enough allowance for the particular _sender.
    - **Impact**: The amount of tokens to be transferred.

Branches and code coverage (including function calls)

Intended branches

- Assumes no malicious intent on behalf of the handler.
  ☑ Test coverage
- If msg.sender is the handler, do not check allowance.
  ☑ Test coverage
- Checks that the _sender has enough balance. Ensured in _transfer.
  ☑ Test coverage
- Checks that the _recipient is not the zero address. Ensured in _transfer.
  ☑ Test coverage
- Checks that the _amount is greater than zero. Ensured in _transfer.
  ☑ Test coverage
- Decreases the balances for the _sender. Ensured in _transfer.
  ☑ Test coverage
- Increases the balances for the _recipient. Ensured in _transfer.
  ☑ Test coverage
- Decreases the allowance of _sender for msg.sender by _amount. Ensured in _approve.
  ☑ Test coverage
- Ensures enough allowance for msg.sender has been granted by _sender.
  ☑ Test coverage

Negative behavior

- Should not allow tranferring more tokens than the _sender has, even if the allowance is greater than the balance.

- ☑ Negative test

- Should not allow transferring more tokens than the allowance granted by the `_sender`.
  - ☑ Negative test

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

Performs the transfer of tokens.

Inputs

- `_recipient`
  - **Control**: Fully controlled by the caller.
  - **Constraints**: None.
  - **Impact**: The account that will receive the tokens.
- `_amount`
  - **Control**: Fully controlled by the caller.
  - **Constraints**: Checked that `msg.sender` can afford to transfer `_amount` of tokens.
  - **Impact**: The amount of tokens to be transferred.

Branches and code coverage (including function calls)

Intended branches

- Checks that the `_sender` has enough balance.
  - ☑ Test coverage
- Checks that the `_recipient` is not the zero address.
  - ☑ Test coverage
- Checks that the `_amount` is greater than zero.
  - ☑ Test coverage
- Decreases the `balances` for the `_sender`.
  - ☑ Test coverage
- Increases the `balances` for the `_recipient`.
  - ☑ Test coverage

Negative behavior

- Should not allow transferring more tokens than the `_sender` has.
  - ☑ Negative test

Function: `unstakeForAccount(address _account, address _receiver, uint256 _depositToken, address _amount)`

Allows unstaking of tokens on behalf of another account.

Inputs

- `_account`
  - **Control**: Fully controlled by the caller (handler).
  - **Constraints**: Checked that `_account` can afford to unstake `_amount` of `_depositToken` (via `depositBalances`, `stakedAmounts`, etc.).
  - **Impact**: The account that will be unstaked for.
- `_receiver`
  - **Control**: Fully controlled by the caller (handler).
  - **Constraints**: None.
  - **Impact**: The beneficiary of the deposit tokens resulting from the unstaking.
- `_depositToken`
  - **Control**: Fully controlled by the caller (handler).
  - **Constraints**: Checked that the token is whitelisted (`isDepositToken`).
  - **Impact**: The token to be unstaked.
- `_amount`
  - **Control**: Fully controlled by the caller (handler).
  - **Constraints**: Checked that the amount is greater than zero. Also, in `safeTransfer` call, it is checked that the `_account` has enough balance.
  - **Impact**: The amount of tokens to be unstaked.

Branches and code coverage (including function calls)

Intended branches

- Assumes no malicious intent on behalf of the handler.
  - ☑ Test coverage
- Checks that the token is whitelisted (`isDepositToken`).
  - ☑ Test coverage
- Checks that the amount is greater than zero.
  - ☑ Test coverage
- Checks that the `_account` has enough balance (`safeTransfer`) — also the check in `depositBalances`.
  - ☑ Test coverage
- Decreases the total deposit supply for the `_depositToken`.
  - ☑ Test coverage

- Decreases the `depositBalances` for the `_account`.
  - ☑ Test coverage
- Decreases the `stakedAmounts` for the `_account`.
  - ☑ Test coverage
- Decreases the `balances` for the `_account` (by burning this contract's tokens).
  - ☑ Test coverage
- Transfers the `_amount` of deposit tokens back to the `_account`.
  - ☑ Test coverage

Negative behavior

- Should not be callable by anyone other than a handler.
  - ☑ Negative test
- Should not allow withdrawing more than the staked amount.
  - ☑ Negative test
- Should not be callable if `isPrivateStakingMode` is `true`.
  - ☑ Negative test
- Assumes user cannot unstake a token that they have not staked. That is handled by the `depositBalances` decrease.
  - ☑ Negative test
- Should not allow unstaking more tokens than the user has staked.
  - ☑ Negative test

Function: `unstake(address _depositToken, uint256 _amount)`

Allows unstaking of tokens.

Inputs

- `_depositToken`
  - **Control**: The `_depositToken` is controlled by the caller.
  - **Constraints**: Checked that the token is whitelisted (`isDepositToken`).
  - **Impact**: The token to be unstaked.
- `_amount`
  - **Control**: The `_amount` is controlled by the caller.
  - **Constraints**: Checked that the amount is greater than zero. Also, in `safeTransfer` call, it is checked that the `_account` has enough balance.
  - **Impact**: The amount of tokens to be unstaked.

Branches and code coverage (including function calls)

Intended branches

- Checks that the token is whitelisted (`isDepositToken`).
  - ☑ Test coverage
- Checks that the amount is greater than zero.
  - ☑ Test coverage
- Checks that the `_account` has enough balance (`safeTransfer`) — also the check in `depositBalances`.
  - ☑ Test coverage
- Decreases the total deposit supply for the `_depositToken`.
  - ☑ Test coverage
- Decreases the `depositBalances` for the `_account`.
  - ☑ Test coverage
- Decreases the `stakedAmounts` for the `_account`.
  - ☑ Test coverage
- Decreases the `balances` for the `_account` (by burning this contract's tokens).
  - ☑ Test coverage
- Transfers the `_amount` of deposit tokens back to the `_account`.
  - ☑ Test coverage

Negative behavior

- Should not allow withdrawing more than the staked amount.
  - ☑ Negative test
- Should not be callable if `isPrivateStakingMode` is `true`.
  - ☑ Negative test
- Assumes user cannot unstake a token that they have not staked. That is handled by the `depositBalances` decrease.
  - ☑ Negative test
- Should not allow unstaking more tokens than the user has staked.
  - ☑ Negative test

Function: `_approve(address _owner, address _spender, uint256 _amount)`

This is the internal function for approving allowances for transferring tokens.

Inputs

- `_owner`
  - **Control**: Controlled by the calling function.
  - **Constraints**: Checked that `_owner` is not the zero address.
  - **Impact**: The account that will allow the `_spender` to transfer tokens.
- `_spender`
  - **Control**: Controlled by the calling function.

  - **Constraints**: Checked that `_spender` is not the zero address.
  - **Impact**: The account that will be allowed to transfer the owner's tokens.
- `_amount`
  - **Control**: Controlled by the calling function.
  - **Constraints**: None.
  - **Impact**: The amount of tokens to be allowed to be transferred.

Branches and code coverage (including function calls)

Intended branches

- Sets the allowance of `_spender` to `_amount` for the `_owner`.
  - ☑ Test coverage

Negative behavior

- Should not allow approving tokens on behalf of others. This should be ensured in the calling function.
  - ☑ Negative test

Function: `_transfer(address _sender, address _recipient, uint256 _amount)`

This is the internal function for transferring tokens.

Inputs

- `_sender`
  - **Control**: Controlled by calling function.
  - **Constraints**: Checked that `_sender` is not the zero address and that it has enough balance.
  - **Impact**: The account that will send the tokens.
- `_recipient`
  - **Control**: Controlled by calling function.
  - **Constraints**: Checked that `_recipient` is not the zero address.
  - **Impact**: The account that will receive the tokens.
- `_amount`
  - **Control**: Controlled by calling function.
  - **Constraints**: Checked that `_sender` has enough balance.
  - **Impact**: The amount of tokens to be transferred.

Branches and code coverage (including function calls)

Intended branches

- Checks that the `_sender` has enough balance.
  - ☑ Test coverage
- Checks that the `_sender` is not the zero address.
  - ☑ Test coverage
- Checks that the `_recipient` is not the zero address.
  - ☑ Test coverage
- Checks that the `_amount` is greater than zero.
  - ☑ Test coverage
- Decreases the `balances` for the `_sender`.
  - ☑ Test coverage
- Increases the `balances` for the `_recipient`.
  - ☑ Test coverage
- Checks that `inPrivateTransferMode` is `true` and calls `_validateHandler`.
  - ☑ Test coverage

Negative behavior

- Should not allow transferring more tokens than the `_sender` has.
  - ☑ Negative test
- Should not allow transferring to the zero address.
  - ☑ Negative test

5.4 Module: StakingAdmin.sol

Function: `execute(address _stakingContract, byte[] _data)`

Executes an arbitrary function.

Inputs

- `_stakingContract`
  - **Control**: Full.
  - **Constraints**: Function must support the staking interfaces.
  - **Impact**: The target contract to be arbitrarily called.
- `_data`
  - **Control**: Full.
  - **Constraints**: None.
  - **Impact**: The data.

Branches and code coverage (including function calls)

Intended branches

- Verifies the returned result.
  - ☑ Test coverage

Negative behavior

- Ensure that functions without the supported interfaces cannot be called
  - ☑ Negative test

Function call analysis

- `execute → _stakingContract.supportsInterface(type(IRewardTracker).interfaceId)`
  - **What is controllable?** `_stakingContract`.
  - **What happens if it reverts, reenters, or does other unusual control flow?** N/A.
  - **If return value is controllable, how is it used and how can it go wrong**: Used to determine if the contract is one of the correct types. (Also stops arbitrary calls to random tokens, in case of compromise.)

Function: `setupGsStaking()`

Sets up all the GS Staking contracts.

Branches and code coverage (including function calls)

Intended branches

- Checks that the handlers are set. (All the functions are callable by the handlers.)
  - ☑ Test coverage

Function call analysis

- `setupGsStaking → IRewardTracker(_rewardTracker).setHandler(_bonusTracker, true)`
  - **What is controllable?** Nothing.
  - **What happens if it reverts, reenters, or does other unusual control flow?** N/A.
  - **If return value is controllable, how is it used and how can it go wrong**: Discarded.
- `setupGsStaking → IRewardTracker(_bonusTracker).setInPrivateClaimingMode`

`(true)`
  - **What is controllable?** Nothing.
  - **What happens if it reverts, reenters, or does other unusual control flow?** N/A.
  - **If return value is controllable, how is it used and how can it go wrong**: Discarded.

Function: `_combineTrackerDistributor(string _name, string _symbol, address _rewardToken, address[] _depositTokens, uint16 _refId, bool _isFeeTracker, bool _isBonusDistributor)`

Sets up a tracker/distributor combo.

Inputs

- `_name`
  - **Control**: Full.
  - **Constraints**: None.
  - **Impact**: Name of the tracker.
- `_symbol`
  - **Control**: Full.
  - **Constraints**: None.
  - **Impact**: Symbol of the tracker.
- `_rewardToken`
  - **Control**: Full.
  - **Constraints**: None.
  - **Impact**: The reward token of the tracker/distributor.
- `_depositTokens`
  - **Control**: Full.
  - **Constraints**: None.
  - **Impact**: The deposit token.
- `_refId`
  - **Control**: Full.
  - **Constraints**: None.
  - **Impact**: The ID.
- `_isFeeTracker`
  - **Control**: Full.
  - **Constraints**: None.
  - **Impact**: Is it the fee tracker (different numbers).
- `_isBonusDistributor`

  - **Control**: Full.
  - **Constraints**: None.
  - **Impact**: Is it the bonus tracker (different numbers).

Branches and code coverage (including function calls)

Intended branches

- Sets the requisite handlers for the deployed contracts.
  - ☑ Test coverage

Negative behavior

- Calls the right deployers for the different types of trackers.
  - ☑ Negative test

5.5 Module: StakingRouter.sol

Function: `claimPool(address _gsPool)`

Allows `msg.sender` to claim all rewards for a pool.

Inputs

- `_gsPool`
  - **Control**: Fully controlled by the handler.
  - **Constraints**: None.
  - **Impact**: The pool to claim from.

Branches and code coverage (including function calls)

Intended branches

- Allows `msg.sender` to claim all rewards for a pool (`rewardTracker`, `loanRewardTracker`, and `vester`).
  - ☑ Test coverage
- Assumes all other relevant checks are performed in each `claimForAccount` function.
  - ☑ Test coverage

Negative behavior

- Should not allow calling an unregistered pool.
  - ☐ Negative test

Function: `claim()`

Allows `msg.sender` to claim all rewards.

Branches and code coverage (including function calls)

Intended branches

- Allows `msg.sender` to claim all rewards.
  ☑ Test coverage
- All underlying claim functions are called (`feeTracker`, `rewardTracker`, `loanRewardTracker`, `vester`, and `loanVester`).
  ☑ Test coverage
- Assumes all other relevant checks are performed in each `claimForAccount` function.
  ☑ Test coverage
- The `bonusTracker` is not called.
  ☐ Test coverage

Function: `compoundForAccount(address _account)`

Allows the handler to compound rewards for an account.

Inputs

- `_account`
  - **Control**: Fully controlled by the handler.
  - **Constraints**: None.
  - **Impact**: Account to compound for.

Branches and code coverage (including function calls)

Intended branches

- Should claim rewards and stake them back.
  ☑ Test coverage
- Assumes all other relevant checks are performed in each `claimForAccount` and `stakeForAccount` function.
  ☑ Test coverage

Negative behavior

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

Function: `compound()`

Allows a user to compound their rewards (i.e., claim -> stake).

Branches and code coverage (including function calls)

Intended branches

- Should claim rewards and stake them back.
  - ☑ Test coverage
- Assumes all other relevant checks are performed in each `claimForAccount` and `stakeForAccount` function.
  - ☑ Test coverage

Function: `stakeEsGsb(uint256 _amount)`

Allows `msg.sender` to stake `esGsb` on behalf of `msg.sender`.

Inputs

- `_amount`
  - **Control**: Fully controlled by the handler.
  - **Constraints**: None.
  - **Impact**: Amount to be staked.

Branches and code coverage (including function calls)

Intended branches

- Allows `msg.sender` to stake `esGsb`.
  - ☑ Test coverage
- Calls `IRewardTracker.stakeForAccount` with `esGsb` tracker.
  - ☑ Test coverage
- Calls `IRewardTracker.stakeForAccount` with `bonus` tracker.
  - ☑ Test coverage
- Calls `IRewardTracker.stakeForAccount` with `fee` tracker.
  - ☑ Test coverage
- Assumes all other relevant checks are performed in `IRewardTracker.stakeForAccount`.
  - ☑ Test coverage

Function: `stakeEsGs(uint256 _amount)`

Allows `msg.sender` to stake `esGs` on behalf of `msg.sender`.

Inputs

- `_amount`
  - **Control**: Fully controlled by the handler.
  - **Constraints**: None.
  - **Impact**: Amount to be staked.

Branches and code coverage (including function calls)

Intended branches

- Allows `msg.sender` to stake `esGs`.
  - [x] Test coverage
- Call `IRewardTracker.stakeForAccount` with `esGs` tracker.
  - [x] Test coverage
- Call `IRewardTracker.stakeForAccount` with `bonus` tracker.
  - [x] Test coverage
- Call `IRewardTracker.stakeForAccount` with `fee` tracker.
  - [x] Test coverage
- Assumes all other relevant checks are performed in `IRewardTracker.stakeForAccount`.
  - [x] Test coverage

Function: `stakeGsForAccount(address _account, uint256 _amount)`

Allows the handler to stake `gs` on behalf of `account`.

Inputs

- `_account`
  - **Control**: Fully controlled by the handler.
  - **Constraints**: None.
  - **Impact**: `gs` is staked on behalf of `_account`.
- `_amount`
  - **Control**: Fully controlled by the handler.
  - **Constraints**: None.
  - **Impact**: `gs` is staked on behalf of `_account`.

Branches and code coverage (including function calls)

Intended branches

- Staked on behalf of the handler for account.
  - ☑ Test coverage
- Calls `IRewardTracker.stakeForAccount` with `gs` tracker.
  - ☑ Test coverage
- Calls `IRewardTracker.stakeForAccount` with `bonus` tracker.
  - ☑ Test coverage
- Calls `IRewardTracker.stakeForAccount` with `fee` tracker.
  - ☑ Test coverage
- Assumes all other necessary checks are performed in `IRewardTracker.stakeForAccount`.
  - ☑ Test coverage

Negative behavior

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

Function: `stakeGs(uint256 _amount)`

Allows `msg.sender` to stake `gs` on behalf of `msg.sender`.

Inputs

- `_amount`
  - **Control**: Fully controlled by the handler.
  - **Constraints**: None.
  - **Impact**: Amount to be staked.

Branches and code coverage (including function calls)

Intended branches

- Allows `msg.sender` to stake `gs`.
  - ☑ Test coverage
- Calls `IRewardTracker.stakeForAccount` with `gs` tracker.
  - ☑ Test coverage
- Calls `IRewardTracker.stakeForAccount` with `bonus` tracker.
  - ☑ Test coverage
- Calls `IRewardTracker.stakeForAccount` with `fee` tracker.
  - ☑ Test coverage

- Assumes all other relevant checks are performed in `IRewardTracker.stakeForAccount`.
  - ☑ Test coverage

Function: `stakeLoanForAccount(address _account, address _gsPool, uint256 _loanId)`

Allows the handler to stake `loan` on behalf of `account`.

Inputs

- `_account`
  - **Control**: Fully controlled by the handler.
  - **Constraints**: None.
  - **Impact**: The `loan` is staked on behalf of `_account`.
- `_gsPool`
  - **Control**: Fully controlled by the handler.
  - **Constraints**: None — should be a registered pool.
  - **Impact**: The pool to stake in.
- `_loanId`
  - **Control**: Fully controlled by the handler.
  - **Constraints**: None.
  - **Impact**: The loan to stake.

Branches and code coverage (including function calls)

Intended branches

- Stake on behalf of the account for account.
  - ☑ Test coverage
- Assumes all other relevant checks are performed in `ILoanTracker.stakeForAccount`.
  - ☑ Test coverage

Negative behavior

- Should not allow calling an unregistered pool.
  - ☐ Negative test
- Should not allow anyone other than the handler to call this function.
  - ☑ Negative test

Function: `stakeLoan(address _gsPool, uint256 _loanId)`

Allows `msg.sender` to stake `loan` on behalf of `msg.sender`.

Inputs

- `_gsPool`
  - **Control**: Fully controlled by the handler.
  - **Constraints**: None — should be a registered pool.
  - **Impact**: The pool to stake in.
- `_loanId`
  - **Control**: Fully controlled by the handler.
  - **Constraints**: None.
  - **Impact**: The loan to stake.

Branches and code coverage (including function calls)

Intended branches

- Allows `msg.sender` to stake `loan`.
  - ☑ Test coverage
- Assumes all other relevant checks are performed in `ILoanTracker.stakeForAccount`.
  - ☑ Test coverage

Negative behavior

- Should not allow calling an unregistered pool.
  - ☐ Negative test

Function: `stakeLpForAccount(address _account, address _amount, uint256 _gsPool)`

Allows handler to stake `lp` on behalf of `account`.

Inputs

- `_account`
  - **Control**: Fully controlled by the handler.
  - **Constraints**: None.
  - **Impact**: `lp` is staked on behalf of `_account`.
- `_amount`
  - **Control**: Fully controlled by the handler.

  - **Constraints**: None.
  - **Impact**: `lp` is staked on behalf of `_account`.
- `_gsPool`
  - **Control**: Fully controlled by the handler.
  - **Constraints**: None — should be a registered pool.
  - **Impact**: The pool to stake in.

Branches and code coverage (including function calls)

Intended branches

- Stake on behalf of `address(this)` for account.
  ☑ Test coverage
- Call `IRewardTracker.stakeForAccount` with `lp` tracker.
  ☑ Test coverage
- Call `IRewardTracker.stakeForAccount` with `bonus` tracker.
  ☑ Test coverage
- Call `IRewardTracker.stakeForAccount` with `fee` tracker.
  ☑ Test coverage
- Assumes all other relevant checks are performed in `IRewardTracker.stakeForAccount`.
  ☑ Test coverage

Negative behavior

- Should not be callable by anyone other than the handler.
  ☑ Negative test
- Should not allow calling an unregistered pool.
  ☐ Negative test

Function: `stakeLp(address _amount, uint256 _gsPool)`

Allows `msg.sender` to stake `lp` on behalf of `msg.sender`.

Inputs

- `_amount`
  - **Control**: Fully controlled by the handler.
  - **Constraints**: None.
  - **Impact**: Amount to be staked.
- `_gsPool`
  - **Control**: Fully controlled by the handler.

        - **Constraints**: None — should be a registered pool.
        - **Impact**: The pool to stake in.

Branches and code coverage (including function calls)

Intended branches

- Allows `msg.sender` to stake `lp`.
  - ☑ Test coverage
- Calls `IRewardTracker.stakeForAccount` with `lp` tracker.
  - ☑ Test coverage
- Calls `IRewardTracker.stakeForAccount` with `bonus` tracker.
  - ☑ Test coverage
- Calls `IRewardTracker.stakeForAccount` with `fee` tracker.
  - ☑ Test coverage
- Assumes all other relevant checks are preformed in `IRewardTracker.stakeForAccount`.
  - ☑ Test coverage

Negative behavior

- Should not allow calling an unregistered pool.
  - ☐ Negative test

Function: `unstakeEsGsb(uint256 _amount)`

Allows `msg.sender` to unstake `esGsb`.

Inputs

- `_amount`
  - **Control**: Fully controlled by the handler.
  - **Constraints**: None.
  - **Impact**: Amount to be unstaked.

Branches and code coverage (including function calls)

Intended branches

- Allows `msg.sender` to unstake `esGsb`.
  - ☑ Test coverage
- All underlying unstake functions are called (`feeTracker`, `bonusTracker`, and `rewardTracker`).

- ☑ Test coverage
- If `shouldReduceBnGs` is true, should increase the amount of `bnGs` staked by `msg.sender`.
  - ☑ Test coverage
- Assumes all other relevant checks are performed in `IRewardTracker.unstakeForAccount`, `IRewardTracker.claimForAccount`, and `IRestrictedToken.burn`.
  - ☑ Test coverage

Negative behavior

- Should not allow unstaking more than the staked amount.
  - ☑ Negative test
- Should not allow unstaking zero amount.
  - ☑ Negative test

Function: `unstakeEsGs(uint256 _amount)`

Allows `msg.sender` to unstake `esGs`.

Inputs

- `_amount`
  - **Control**: Fully controlled by handler.
  - **Constraints**: None.
  - **Impact**: Amount to be unstaked.

Branches and code coverage (including function calls)

Intended branches

- Allows `msg.sender` to unstake `esGs`.
  - ☑ Test coverage
- All underlying unstake functions are called (`feeTracker`, `bonusTracker`, and `rewardTracker`).
  - ☑ Test coverage
- If `shouldReduceBnGs` is true, should increase the amount of `bnGs` staked by `msg.sender`.
  - ☑ Test coverage
- Assumes all other relevant checks are performed in `IRewardTracker.unstakeForAccount`, `IRewardTracker.claimForAccount`, and `IRestrictedToken.burn`.
  - ☑ Test coverage

Negative behavior

- Should not allow unstaking more than the staked amount.
  - ☑ Negative test
- Should not allow unstaking zero amount.
  - ☑ Negative test

Function: `unstakeGs(uint256 _amount)`

Allows `msg.sender` to unstake `gs`.

Inputs

- `_amount`
  - **Control**: Fully controlled by the handler.
  - **Constraints**: None.
  - **Impact**: Amount to be unstaked.

Branches and code coverage (including function calls)

Intended branches

- Allows `msg.sender` to unstake `gs`.
  - ☑ Test coverage
- All underlying unstake functions are called (`feeTracker`, `bonusTracker`, and `rewardTracker`).
  - ☑ Test coverage
- If `shouldReduceBnGs` is true, should increase the amount of `bnGs` staked by `msg.sender`.
  - ☑ Test coverage
- Assumes all other relevant checks are performed in `IRewardTracker.unstakeForAccount`, `IRewardTracker.claimForAccount`, and `IRestrictedToken.burn`.
  - ☑ Test coverage

Negative behavior

- Should not allow unstaking more than the staked amount.
  - ☑ Negative test
- Should not allow unstaking zero amount.
  - ☑ Negative test

Function: `unstakeLoanForAccount(address _account, address _gsPool, uint256 _amount)`

Allows the handler to unstake `loan` on behalf of `account`.

Inputs

- `_account`
  - **Control**: Fully controlled by the handler.
  - **Constraints**: None.
  - **Impact**: `loan` is unstaked on behalf of `_account`.
- `_gsPool`
  - **Control**: Fully controlled by the handler.
  - **Constraints**: None — should be a registered pool.
  - **Impact**: The pool to unstake from.
- `_amount`
  - **Control**: Fully controlled by the handler.
  - **Constraints**: None.
  - **Impact**: Amount to be unstaked.

Branches and code coverage (including function calls)

Intended branches

- Unstakes from `gsPool` for `account`'s reward tracker.
  - ☑ Test coverage
- Acts on behalf of `_account`.
  - ☑ Test coverage
- Assumes all other relevant checks are performed in `ILoanTracker.unstakeForAccount`.
  - ☑ Test coverage
- Should not allow calling an unregistered pool.
  - ☐ Test coverage

Negative behavior

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

Function: `unstakeLoan(address _gsPool, uint256 _amount)`

Allows `msg.sender` to unstake `loan`.

Inputs

- `_gsPool`
  - **Control**: Fully controlled by the handler.
  - **Constraints**: None — should be a registered pool.

  - **Impact**: The pool to unstake from.
- `_amount`
  - **Control**: Fully controlled by the handler.
  - **Constraints**: None.
  - **Impact**: Amount to be unstaked.

Branches and code coverage (including function calls)

Intended branches

- Unstakes from `gsPool` for `account`'s reward tracker.
  - ☑ Test coverage
- Assumes all other relevant checks are performed in `ILoanTracker.unstakeForAccount`.
  - ☑ Test coverage
- Should not allow calling an unregistered pool.
  - ☐ Test coverage

Negative behavior

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

Function: `unstakeLpForAccount(address _account, address _gsPool, uint256 _amount)`

Allows handler to unstake `lp` on behalf of `account`.

Inputs

- `_account`
  - **Control**: Fully controlled by the handler.
  - **Constraints**: None.
  - **Impact**: `lp` is unstaked on behalf of `_account`.
- `_gsPool`
  - **Control**: Fully controlled by the handler.
  - **Constraints**: None; should be a registered pool.
  - **Impact**: The pool to unstake from.
- `_amount`
  - **Control**: Fully controlled by the handler.
  - **Constraints**: None.
  - **Impact**: Amount to be unstaked.

Branches and code coverage (including function calls)

Intended branches

- Unstakes from `gsPool` for `account`'s reward tracker.
  - ☑ Test coverage
- Acts on behalf of `_account`.
  - ☑ Test coverage
- Assumes all other relevant checks are performed in `IRewardTracker.unstakeForAccount`.
  - ☑ Test coverage

Negative behavior

- Should not allow calling an unregistered pool.
  - ☐ Negative test
- Should not be callable by anyone other than the handler.
  - ☑ Negative test

Function: `unstakeLp(address _gsPool, uint256 _amount)`

Allows `msg.sender` to unstake `lp`.

Inputs

- `_gsPool`
  - **Control**: Fully controlled by the handler.
  - **Constraints**: None — should be a registered pool.
  - **Impact**: The pool to unstake from.
- `_amount`
  - **Control**: Fully controlled by handler.
  - **Constraints**: None.
  - **Impact**: Amount to be unstaked.

Branches and code coverage (including function calls)

Intended branches

- Unstakes from `gsPool` for `account`'s reward tracker.
  - ☑ Test coverage
- Assumes all other relevant checks are performed in `IRewardTracker.unstakeForAccount`.
  - ☑ Test coverage

- Should not allow calling an unregistered pool.
  ☐ Test coverage

Negative behavior

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

Function: `vestEsGsForPool(address _gsPool, uint256 _amount)`

Allows vesting of `esGs` for a pool.

Inputs

- `_gsPool`
  - **Control**: Fully controlled by the handler.
  - **Constraints**: None.
  - **Impact**: The pool to vest in.
- `_amount`
  - **Control**: Fully controlled by the handler.
  - **Constraints**: None.
  - **Impact**: Amount to be vested.

Inputs

- `_amount`
  - **Control**: Fully controlled by the handler.
  - **Constraints**: None.
  - **Impact**: Amount to be vested.

Branches and code coverage (including function calls)

Intended branches

- Allows vesting of `esGs` through `IVester.depositForAccount` in a pool.
  ☑ Test coverage
- Assumes all other relevant checks are performed in `IVester.depositForAccount`.
  ☑ Test coverage

Negative behavior

- Should not allow calling an unregistered pool.
  ☐ Negative test

Function: `vestEsGsb(uint256 _amount)`

Allows vesting of `esGsb`.

Inputs

- `_amount`
  - **Control**: Fully controlled by the handler.
  - **Constraints**: None.
  - **Impact**: Amount to be vested.

Branches and code coverage (including function calls)

Intended branches

- Allows vesting of `esGsb` through `IVester.depositForAccount`.
  ☑ Test coverage
- Assumes all other relevant checks are preformed in `IVester.depositForAccount`
  ☑ Test coverage

Function: `vestEsGs(uint256 _amount)`

Allows vesting of `esGs`.

Inputs

- `_amount`
  - **Control**: Fully controlled by the handler.
  - **Constraints**: None.
  - **Impact**: Amount to be vested.

Branches and code coverage (including function calls)

Intended branches

- Allows vesting of `esGs` through `IVester.depositForAccount`.
  ☑ Test coverage
- Assumes all other relevant checks are performed in `IVester.depositForAccount`
  ☑ Test coverage

Function: `withdrawEsGsForPool(address _gsPool)`

Allows `msg.sender` to withdraw `esGs` for a pool.

Inputs

- `_gsPool`
  - **Control**: Fully controlled by the handler.
  - **Constraints**: None.
  - **Impact**: The pool to withdraw from.

Branches and code coverage (including function calls)

Intended branches

- Allows `msg.sender` to withdraw `esGs` through `IVester.withdrawForAccount` in a pool.
  - ☑ Test coverage
- Assumes all other relevant checks are performed in `IVester.withdrawForAccount`.
  - ☑ Test coverage

Negative behavior

- Should not allow calling an unregistered pool.
  - ☐ Negative test

Function: `withdrawEsGsb()`

Allows `msg.sender` to withdraw `esGsb`.

Branches and code coverage (including function calls)

Intended branches

- Allows `msg.sender` to withdraw `esGsb` through `IVester.withdrawForAccount`.
  - ☑ Test coverage
- Assumes all other relevant checks are performed in `IVester.withdrawForAccount`.
  - ☑ Test coverage

Function: `withdrawEsGs()`

Allows `msg.sender` to withdraw `esGs`.

Branches and code coverage (including function calls)

Intended branches

- Allows `msg.sender` to withdraw `esGs` through `IVester.withdrawForAccount`.
  ☑ Test coverage
- Assumes all other relevant checks are performed in `IVester.withdrawForAccount`.
  ☑ Test coverage

5.6 Module: VesterNoReserve.sol

Function: `claimable(address _account)`

Returns the amount of tokens that are done with vesting and are claimable for a specific account.

Inputs

- `_account`
  - **Control**: Full.
  - **Constraints**: None.
  - **Impact**: The account.

Branches and code coverage (including function calls)

Intended branches

- Returns the claimable amount.
  ☑ Test coverage

Negative behavior

- Does not return the same amount after a claim.
  ☑ Negative test

Function: `claim()`

Claims the vested tokens of `msg.sender`.

Branches and code coverage (including function calls)

Intended branches

- Tokens are claimed.
  ☑ Test coverage

Function call analysis

- 'claim() -> _updateVesting(...) -> _burn(account, amount)
  - **What is controllable?** Nothing.
  - **What happens if it reverts, reenters, or does other unusual control flow?** N/A.
  - **If return value is controllable, how is it used and how can it go wrong**: Discarded.
- `claim() → _updateVesting( ... ) → IRestrictedToken(esToken).burn(_account, amount)`
  - **What is controllable?** Nothing.
  - **What happens if it reverts, reenters, or does other unusual control flow?** User does not have enough tokens.
  - **If return value is controllable, how is it used and how can it go wrong**: Discarded.
- `claim() → _claim( ... ) → claimableToken.safeTransfer(_receiver, amount)`
  - **What is controllable?** `_receiver`.
  - **What happens if it reverts, reenters, or does other unusual control flow?** Nothing.
  - **If return value is controllable, how is it used and how can it go wrong**: Discarded.

Function: `deposit(uint256 _amount)`

Deposits and vests tokens.

Inputs

- `_amount`
  - **Control**: Full.
  - **Constraints**: None.
  - **Impact**: Amount of tokens.

Branches and code coverage (including function calls)

Intended branches

- Deposits the user's tokens and mints them the vested equivalent.
  ☑ Test coverage

Negative behavior

- Does not allow user to vest more than the max vestable amount.

☑ Negative test

Function call analysis

- `deposit() → _deposit() → _updateVesting( … ) → _burn(account, amount)`
  - **What is controllable?** Nothing.
  - **What happens if it reverts, reenters, or does other unusual control flow?** N/A.
  - **If return value is controllable, how is it used and how can it go wrong**: Discarded.
- `deposit() → _deposit() → _updateVesting( … ) → IRestrictedToken(esToken).burn(_account, amount)`
  - **What is controllable?** Nothing.
  - **What happens if it reverts, reenters, or does other unusual control flow?** User does not have enough tokens.
  - **If return value is controllable, how is it used and how can it go wrong**: Discarded.
- `deposit() → _deposit() → esToken.safeTransferFrom(_account, address(this), _amount)`
  - **What is controllable?** Amount.
  - **What happens if it reverts, reenters, or does other unusual control flow?** User does not have enough tokens.
  - **If return value is controllable, how is it used and how can it go wrong**: Discarded.
- `deposit() → _deposit() → getMaxVestableAmount → IRewardTracker(rewardTracker).cumulativeRewards(_account)`
  - **What is controllable?** Amount.
  - **What happens if it reverts, reenters, or does other unusual control flow?** User does not have enough tokens.
  - **If return value is controllable, how is it used and how can it go wrong**: Discarded.

Function: `withdraw()`

Ends the vesting early and withdraws.

Branches and code coverage (including function calls)

Intended branches

- Returns the user their deposit.
  ☑ Test coverage

Negative behavior

- Does not return the user any bonuses.
  - ☑ Negative test

Function call analysis

- `withdraw() → _withdraw( … ) → esToken.safeTransfer(_account, balance)`
  - **What is controllable?** Nothing.
  - **What happens if it reverts, reenters, or does other unusual control flow?** N/A.
  - **If return value is controllable, how is it used and how can it go wrong**: Discarded.
- `withdraw() → _withdraw( … ) → esToken_burn(_account, balance)`
  - **What is controllable?** Nothing.
  - **What happens if it reverts, reenters, or does other unusual control flow?** N/A.
  - **If return value is controllable, how is it used and how can it go wrong**: Discarded.

5.7 Module: Vester.sol

Function: `claim()`

Claims the vested tokens of `msg.sender`.

Branches and code coverage (including function calls)

Intended branches

- Tokens are claimed.
  - ☑ Test coverage

Function call analysis

- 'claim() -> _updateVesting(...) -> _burn(account, amount)
  - **What is controllable?** Nothing.
  - **What happens if it reverts, reenters, or does other unusual control flow?** N/A.
  - **If return value is controllable, how is it used and how can it go wrong**: Discarded.
- `claim() → _updateVesting( … ) → IRestrictedToken(esToken).burn(_accoun`

`t, amount)`
    - **What is controllable?** Nothing.
    - **What happens if it reverts, reenters, or does other unusual control flow?** User does not have enough tokens.
    - **If return value is controllable, how is it used and how can it go wrong**: Discarded.
- `claim()` → `_claim( ... )` → `claimableToken.safeTransfer(_receiver, amount)`
    - **What is controllable?**: `_receiver`.
    - **What happens if it reverts, reenters, or does other unusual control flow?** Nothing.
    - **If return value is controllable, how is it used and how can it go wrong**: Discarded.

Function: `deposit(uint256 _amount)`

Deposits and vests tokens.

Inputs

- `_amount`
    - **Control**: Full.
    - **Constraints**: None.
    - **Impact**: Amount of tokens.

Branches and code coverage (including function calls)

Intended branches

- Deposits the user's tokens and mints them the vested equivalent.
    - ☑ Test coverage

Negative behavior

- Does not allow user to vest more than the max vestable amount.
    - ☑ Negative test

Function call analysis

- 'deposit() -> _deposit() -> _updateVesting(...) -> _burn(account, amount)
    - **What is controllable?** Nothing.
    - **What happens if it reverts, reenters, or does other unusual control flow?** N/A.
    - **If return value is controllable, how is it used and how can it go wrong**:

Discarded.

- `deposit()` → `_deposit()` → `_updateVesting( ... )` → `IRestrictedToken(esToken).burn(_account, amount)`
  - **What is controllable?** Nothing.
  - **What happens if it reverts, reenters, or does other unusual control flow?** User does not have enough tokens.
  - **If return value is controllable, how is it used and how can it go wrong**: Discarded.
- `deposit()` → `_deposit()` → `esToken.safeTransferFrom(_account, address(this), _amount)`
  - **What is controllable?** Amount.
  - **What happens if it reverts, reenters, or does other unusual control flow?** User does not have enough tokens.
  - **If return value is controllable, how is it used and how can it go wrong**: Discarded.
- `deposit()` → `_deposit()` → `getMaxVestableAmount` → `IRewardTracker(rewardTracker).cumulativeRewards(_account)`
  - **What is controllable?** Amount.
  - **What happens if it reverts, reenters, or does other unusual control flow?** N/A.
  - **If return value is controllable, how is it used and how can it go wrong**: Discarded.
- `deposit()` → `_deposit()` → `getMaxVestableAmount` → `IRewardTracker(rewardTracker).averageStakedAmounts(_account)`
  - **What is controllable?** Nothing.
  - **What happens if it reverts, reenters, or does other unusual control flow?** N/A.
  - **If return value is controllable, how is it used and how can it go wrong**: Discarded.
- `deposit()` → `_deposit()` → `getMaxVestableAmount` → `IRewardTracker(rewardTracker).cumulativeRewards(_account)`
  - **What is controllable?** Amount.
  - **What happens if it reverts, reenters, or does other unusual control flow?** N/A.
  - **If return value is controllable, how is it used and how can it go wrong**: Discarded.

Function: `withdraw()`

Ends the vesting early and withdraws.

Branches and code coverage (including function calls)

Intended branches

- Returns the user their deposit.
  - ☑ Test coverage

Negative behaviour

- Does not return the user any bonuses.
  - ☑ Negative test

Function call analysis

- `withdraw() → _withdraw( … ) → pairToken.safeTransfer(_account, pairAmount)`
  - **What is controllable?** Nothing.
  - **What happens if it reverts, reenters, or does other unusual control flow?** N/A.
  - **If return value is controllable, how is it used and how can it go wrong**: Discarded.
- `withdraw() → _withdraw( … ) → esToken.safeTransfer(_account, balance)`
  - **What is controllable?** Nothing.
  - **What happens if it reverts, reenters, or does other unusual control flow?** N/A.
  - **If return value is controllable, how is it used and how can it go wrong**: Discarded.

6 Assessment Results

At the time of our assessment, the reviewed code was not deployed to the Ethereum Mainnet.

During our assessment on the scoped Staking contracts, we discovered four findings. No critical issues were found. One finding was of high impact, one was of medium impact, and the remaining findings were informational in nature. GammaSwap 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 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.