BAILSEC.IO

OFFICE@BAILSEC.IO

X: @BAILSECURITY

TG: @HELLOATBAILSEC

FINAL REPORT:

zkSwap Finance

April 2024

Disclaimer:

Security assessment projects are time-boxed and often reliant on information that may be provided by a client, its affiliates, or its partners. As a result, the findings documented in this report should not be considered a comprehensive list of security issues, flaws, or defects in the target system or codebase.

The content of this assessment is not an investment. The information provided in this report is for general informational purposes only and is not intended as investment, legal, financial, regulatory, or tax advice. The report is based on a limited review of the materials and documentation provided at the time of the audit, and the audit results may not be complete or identify all possible vulnerabilities or issues. The audit is provided on an "as-is," "where-is," and "as-available" basis, and the use of blockchain technology is subject to unknown risks and flaws.

The audit does not constitute an endorsement of any particular project or team, and we make no warranties, expressed or implied, regarding the accuracy, reliability, completeness, or availability of the report, its content, or any associated services or products. We disclaim all warranties, including the implied warranties of merchantability, fitness for a particular purpose, and non-infringement.

We assume no responsibility for any product or service advertised or offered by a third party through the report, any open-source or third-party software, code, libraries, materials, or information linked to, called by, referenced by, or accessible through the report, its content, and the related services and products. We will not be liable for any loss or damages incurred as a result of the use or reliance on the audit report or the smart contract.

The contract owner is responsible for making their own decisions based on the audit report and should seek additional professional advice if needed. The audit firm or individual assumes no liability for any loss or damages incurred as a result of the use or reliance on the audit report or the smart contract. The contract owner agrees to indemnify and hold harmless the audit firm or individual from any and all claims, damages, expenses, or liabilities arising from the use or reliance on the audit report or the smart contract.

By engaging in a smart contract audit, the contract owner acknowledges and agrees to the terms of this disclaimer.

1. Project Details

Important:
Please ensure that the deployed contract matches the source-code of the last commit hash.

Project	zkSwap Finance
Website	zkswap.finance
Type	Vault
Language	Solidity
Methods	Manual Analysis
Github repository	https://github.com/ZkSwapFinance/governanceStaking-contracts/blob/87d5b83172776c4691cfdf97600045a2a4921432/contracts/ZFGovernanceStaking.sol
Resolution 1	https://github.com/ZkSwapFinance/governanceStaking-contracts/blob/be79f72e486e76fe04f107f702822f3eff64dcd7/contracts/ZFGovernanceStaking.sol

2. Detection Overview

Severity	Found	Resolved	Partially Resolved	Acknowledged (no change made)
High	2	2		
Medium	2	2		
Low	1	1		
Informational	1	1		
Governance	1	1		
Total	7	7		

2.1 Detection Definitions

Severity	Description
High	The problem poses a significant threat to the confidentiality of a considerable number of users' sensitive data. It also has the potential to cause severe damage to the client's reputation or result in substantial financial losses for both the client and the affected users.
Medium	While medium level vulnerabilities may not be easy to exploit, they can still have a major impact on the execution of a smart contract. For instance, they may allow public access to critical functions, which could lead to serious consequences.
Low	Poses a very low-level risk to the project or users. Nevertheless the issue should be fixed immediately
Informational	Effects are small and do not post an immediate danger to the project or users
Governance	Governance privileges which can directly result in a loss of funds or other potential undesired behavior

3. Detection

GovernanceStaking

The GovernanceStaking contract is a staking contract that allows users to stake zfTokens and receive yZFTokens as receipt token, which reflects the staking position.

The vault is meant to be reward accruing, which means that over time, zfTokens are being minted to the vault, which will increase the underlying balance of zfTokens and therefore will also increase the exchange rate and the value of the receipt tokens (yZFToken).

As an illustrated example:

Initially, 1e18 yZFTokens represent 1e18 zfToken and after some time, this will shift towards 1e18 yZFTokens representing 1.1e18 zfTokens.

ZFTokens are minted over time, starting from the startTime time spanning up to the determined endTime. The calculation is based on the zfRewardRate, which represents the rate per second and the time passed since the last update.

Upon stake, users will always receive the corresponding yZFTokens based on the current exchange rate and the provided ZFToken amount, whereas upon unstake, users will receive the corresponding ZFTokens based on the provided amount of yZFTokens and the current exchange ratio. It is ensured that the reward state is always updated before any calculation in stake/unstake.

A penalty mechanism was implemented which incentivizes users to keep their positions in the vault, as upon each unstake, a withdrawal fee is deducted from the received ZFToken amount, which is then distributed amongst several governance determined addresses. If the penaltyList is empty or the sum of the allocations does not cover the withdrawalFee, the leftover of the withdrawalFee is distributed equally among all stake participants.

Disclaimer: The yZFToken, which is inherited by the GovernanceStaking contract is excluded from the scope.

Issue_01	Governance Privilege: Lack of safeguard for multiple sensitive settings
Severity	**Governance**
Description	Currently, governance of this contract has several privileges for invoking certain functions that can drastically alter the contract's behavior. It is important to understand that this issue is separated from the mentioned flaws for the reward setting logic because the flaws described in the other issues, are not dependent on a malbehavior of the contract owner but will inherently happen upon reward adjustment. a) Lack of validation within the constructor: Several parameters lack upper limits and logical validation such as ensuring that the rewardRate is not unreasonably high and the start and end time are reasonably chosen. b) Settings during the normal business logic: Several setter functions, similar as in the constructor, allow the owner to set arbitrary rates, eventually harming the health of the protocol. c) Penalty logic allows to exceed MAX_ALLOCATION_RATIO with aggregated penalty addresses: It is possible to add addresses which will receive a share of the penalty amount whenever a user withdraws. A problem hereby is within the addPenaltyAllocation function, as there is no check which ensures that the aggregate of all penalties in the list is smaller than MAX_ALLOCATION_RATIO. If there is ever such a case that this precision is exceeded, the stakers will bear this loss.
Recommendations	Consider incorporating a Gnosis Multisignature contract as owner and ensuring that the Gnosis participants are trusted entities.

Comments / Resolution	Resolved: Settings within the constructor have been validated and a validation for the allocation threshold has been implemented. It must be noted that the owner may still set the startTime to a value which is smaller than zfLastRewardTime. In that scenario, rewards might be double allocated. Under certain circumstances, it might be even desired to keep this freedom. The ZkSwap team is however a very trustworthy team and ensured that such a misconfiguration will not happen.

Issue_02	Stake function is vulnerable to share inflation attack
Severity	**High**
Description	The stake function is inherently vulnerable to the widely known share inflation attack. This attack leverages the following calculation to induce a beneficial exchange ratio for the first staker: shares = amount * totalSupply / poolBalance; This attack unfolds as follows: a) Alice invokes stake with 1 WEI. b) Alice transfers a specific amount of ZFTokens directly to the contract. The rationale behind this move is to inflate the “poolBalance” value. The higher the value, the more effective the attack. Let’s use 100e18 as an example. c) Bob now invokes stake with any amount, as example 1e18. Due to the inflation of the poolBalance, the share calculation for Bob will be as follows:

	shares = 1e18 * 1 / 100e18 -> 0 d) Bob will receive 0 shares, while he effectively deposited 1e18 into the vault. These 1e18 tokens will be allocated to Alice. Alice effectively stole Bob's deposit.
Recommendations	There are multiple scenarios to safeguard against the share inflation attack, such as burning an initial amount of shares, using OZ's approach of virtual shares or simply requiring the shares to be non-zero. A more trivial, but still effective approach is the following: https://bailsec.io/tpost/vk6pakcjv1-safeguarding-erc4626-vaults-from-inflati
Comments / Resolution	Resolved, a check has been implemented which ensures that shares cannot be zero. Additionally, the client will execute the first deposit.

Issue_03	**pendingZF logic can implicitly and unintentionally result in share value manipulation**
Severity	**High**
Description	Similar to the previous issue, the fact that the vault accrues value before a deposit has happened can also result in an undesired exchange rate manipulation. Consider the following scenario: 1) The vault is deployed and accrues rewards since 86400 seconds (1 day) with a rewardRate of 1e18. No deposit has happened thus far.

	2) Bob stakes 1000 WEI as first staker 3) Alice subsequently stakes 1e18 tokens: 1e18 * 1e4 / (86400e18+1000) = 0 shares 4) Alice will receive no shares while Bob will receive Alice's tokens This issue is - contrary to the previous issue - not based on a malicious action but rather based on an edge-case within the contracts business logic.
Recommendations	In Addition to the previous fix we recommend that the owner executes the very first deposit. This will ensure that a reasonable amount of shares is existent, which mitigates the impact.
Comments / Resolution	Resolved, a check has been implemented which ensures that shares cannot be zero. Additionally, the client will execute the first deposit.

Issue_04	New setting of endTime will distribute all rewards in hindsight
Severity	**Medium**
Description	The setEndTime function allows a privileged address to change the endTime to any arbitrary time. There are two distinct problems with this function, whereas the second is more dramatic: a) The owner can set the endTime to a smaller value than zfLastRewardTime, which effectively will prevent reward distribution from rewards which have not been granted yet, but were expected by users. b) Whenever the current endTime has passed, the contract will not

mint any rewards anymore, which is perfectly fine. This also means that upon any interaction, the zfLastRewardTime is **not** updated anymore.

Consider the scenario where the current endTime was at block.timestamp = 10_000. A user now stakes or withdraws at this timestamp and rewards are minted + zfLastRewardTime is set to 10_000:

https://github.com/ZkSwapFinance/governanceStaking-contracts/blob/87d5b83172776c4691cfdf97600045a2a4921432/contracts/ZFGovernanceStaking.sol#L154

It is important to note that this value is only set, if the return value of pendingZF is > 0:

https://github.com/ZkSwapFinance/governanceStaking-contracts/blob/87d5b83172776c4691cfdf97600045a2a4921432/contracts/ZFGovernanceStaking.sol#L152

Due to the fact that the endTime was reached and future stake/unstake interactions will not result in any minted rewards, the zfLastRewardTime is not altered anymore, even if stake/unstake is invoked, still staying at TS = 10_000.

After some time, let's say we are at TS = 15_000, the contract owner decides to “re-enable” rewards, by simply setting the endTime to 20_000. Theoretically, this means starting from TS 15_000, rewards will start to be distributed again.

This is however incorrect because rewards will be distributed back in hindsight starting from TS = 10_000, effectively distributing tokens which should not be distributed.

With a very high likelihood this will happen during the normal

	business logic at some point and very sophisticated users can frontrun this scenario, which then results in these users gaining an unfair amount of rewards.
Recommendations	Consider updating the rewards and setting zfLastRewardTime to the current block.timestamp whenever the setEndTime function is invoked. The approach for this is as follows: a) Invoke pendingZF b) Mint pending rewards c) Set zfLastRewardTime to the current block.timestamp Additionally, it should be ensured that endTime cannot be set to any value in the past.
Comments / Resolution	Resolved, the endTime variable cannot be changed anymore.

Issue_05	Update of rewardRate will be applied on hindsight distribution
Severity	**Medium**
Description	The contract owner can update the rewardRate via the setRewardRate function. The problem with this function is that it does not execute an update up to the current timestamp but updates the rewardRate optimistically. This will alter the rewardRate in reference to the zfLastRewardTime, distributing a different amount of rewards not only since the current update but since the last time the rewards were updated. Additionally, it needs to be mentioned that if the rewardRate is set to zero, this will inherently result in pendingRewards returning zero. Therefore, the zfLastUpdateTime is never updated during stake/unstake.

	In reverse, this means if the rewardRate is then set from a zero value to a non-zero value, full rewards are granted in hindsight for the whole time which has passed since zfLastUpdateTime.
Recommendations	Consider updating the rewards and setting zfLastRewardTime to the current block.timestamp whenever the rewardRate is changed. The approach for this is as follows: a) Invoke pendingZF b) Mint pending rewards c) Set zfLastRewardTime to the current block.timestamp
Comments / Resolution	Resolved, rewards are now updated before the rate is changed.

Issue_06	Length of penaltyList can result in removal issues
Severity	**Low**
Description	Currently, the contract owner can add penalty allocations via the addPenaltyAllocation function. Upon unstake interactions it will then loop over the penaltyList array. The removal is done via the removePenaltyAllocation function. However, the removal process is very unconventional and will not work due to an out-of-bond access within the loop. This is however no large problem due to the fact that existing addresses' allocations can just be zero-d out. A more elegant approach to remove a distinct element from an array is the swap and pop method, which is explained here: https://bailsec.io/tpost/n9plh6cpo1-swap-and-pop-in-solidity

Recommendations	Consider using the swap and pop method.
Comments / Resolution	Resolved, the swap and pop method was implemented.

Issue_07	Violation of checks-effects-interactions pattern
Severity	**Informational**
Description	Throughout the contract there are one or multiple spots which violate the checks-effects-interactions pattern, to ensure a protection against invalid states, all external calls should strictly be implemented after any checks and effects (state variable changes). L 137: https://github.com/ZkSwapFinance/governanceStaking-contracts/blob/87d5b83172776c4691cfdf97600045a2a4921432/contracts/ZFGovernanceStaking.sol#L137 The external transfer is executed before the _mint function. Contrary to the standard ERC4626 implementation, this is a valid recommendation because the special scenario of the ERC777 token is never given here.
Recommendations	Consider following the cei pattern and executing the _mint before the external transfer. (https://bailsec.io/tpost/gxcih1xoy1-checks-effects-interactions)
Comments / Resolution	Resolved, _mint is now executed before the external transfer.