verichains

SECURITY AUDIT OF

KONVERTER

Public Report

Jan 09, 2023

Verichains Lab

info@verichains.io

https://www.verichains.io

Driving Technology > Forward

ABBREVIATIONS

Name	Description
Ethereum	An open source platform based on blockchain technology to create and distribute smart contracts and decentralized applications.
Ether (ETH)	A cryptocurrency whose blockchain is generated by the Ethereum platform. Ether is used for payment of transactions and computing services in the Ethereum network.
Smart contract	A computer protocol intended to digitally facilitate, verify or enforce the negotiation or performance of a contract.
Solidity	A contract-oriented, high-level language for implementing smart contracts for the Ethereum platform.
Solc	A compiler for Solidity.
ERC20	ERC20 (BEP20 in Binance Smart Chain or *x*RP20 in other chains) tokens are blockchain-based assets that have value and can be sent and received. The primary difference with the primary coin is that instead of running on their own blockchain, ERC20 tokens are issued on a network that supports smart contracts such as Ethereum or Binance Smart Chain.

EXECUTIVE SUMMARY

This Security Audit Report was prepared by Verichains Lab on Jan 09, 2023. We would like to thank the Konverter for trusting Verichains Lab in auditing smart contracts. Delivering high-quality audits is always our top priority.

This audit focused on identifying security flaws in code and the design of the Konverter. The scope of the audit is limited to the source code files provided to Verichains. Verichains Lab completed the assessment using manual, static, and dynamic analysis techniques.

During the audit process, the audit team had identified some vulnerable issues in the smart contracts code.

TABLE OF CONTENTS

1. MANAGEMENT SUMMARY

1.1. About Konverter

Konverter is a tool developed by the Klaytn team at Ground X for converting Klaytn tokens between different formats. Konverter is a tool that allows users to convert their KLAY tokens, the native token of the Klaytn platform, between ERC-20 and BEP2 formats. ERC-20 is a standard for tokens on the Ethereum blockchain, while BEP2 is a standard for tokens on the Binance Chain. By converting their KLAY tokens between these different formats, users can use them on different blockchain platforms and exchanges.

1.2. Audit scope

This audit focused on identifying security flaws in code and the design of the Konverter.

It was conducted on the source code provided by Konverter team. The latest version of the following repositories were made available in the course of the review:

Repository	Commit
https://bitbucket.org/wm-public/konverter-contract-eth-audit	33d22a8c19487b456cf0bf621462fa56d0bc3eee
https://bitbucket.org/wm-public/konverter-contract-klaytn-audit	4046758ca1da9e6e4cbfcd10d8cd7e26da22858e

Table 1. The Konverter repositories

1.3. Audit methodology

Our security audit process for smart contract includes two steps:

- Smart contract codes are scanned/tested for commonly known and more specific vulnerabilities using public and RK87, our in-house smart contract security analysis tool.
- Manual audit of the codes for security issues. The contracts are manually analyzed to look for any potential problems.

Following is the list of commonly known vulnerabilities that were considered during the audit of the smart contract:

- Integer Overflow and Underflow
- Timestamp Dependence
- Race Conditions
- Transaction-Ordering Dependence

- DoS with (Unexpected) revert
- DoS with Block Gas Limit
- Gas Usage, Gas Limit and Loops
- Redundant fallback function
- Unsafe type Inference
- Reentrancy
- Explicit visibility of functions state variables (external, internal, private and public)
- Logic Flaws

For vulnerabilities, we categorize the findings into categories as listed in table below, depending on their severity level:

SEVERITY LEVEL	DESCRIPTION
CRITICAL	A vulnerability that can disrupt the contract functioning; creates a critical risk to the contract; required to be fixed immediately.
HIGH	A vulnerability that could affect the desired outcome of executing the contract with high impact; needs to be fixed with high priority.
MEDIUM	A vulnerability that could affect the desired outcome of executing the contract with medium impact in a specific scenario; needs to be fixed.
LOW	An issue that does not have a significant impact, can be considered as less important.

Table 2. Severity levels

1.4. Disclaimer

Please note that security auditing cannot uncover all existing vulnerabilities, and even an audit in which no vulnerabilities are found is not a guarantee for a 100% secure smart contract. However, auditing allows discovering vulnerabilities that were unobserved, overlooked during development and areas where additional security measures are necessary.

2. AUDIT RESULT

2.1. Overview

The Konverter was written in `Solidity` language, with the required version to be `^0.8.9`. The source code was written based on OpenZeppelin's library.

2.1.1. konverter-contract-eth-audit

KonverterLocker.sol

The KonverterLocker contract appears to be a contract that allows users to lock and unlock their Klaytn (KVT) tokens. The contract includes a function called "lock" which allows users to transfer their KVT tokens from their own wallet to the contract. The contract also includes a function called "unLock" which allows users to transfer their KVT tokens from the contract back to their own wallet. The "unLock" function is restricted to only allow users who are on the contract's whitelist to execute the function.

The contract also includes a state variable called "totalLocked" which keeps track of the total amount of KVT tokens that are currently locked in the contract.

KonverterMinter.sol

The KonverterMinter is a contract that allows users to mint new Klaytn (KVT) tokens and lock them in a separate contract called the "locker" contract. The contract includes a function called "mint" which allows users in whitelist to mint new KVT tokens and transfer them to a specified address. The contract also includes a function called "mintAndLock" which allows users in whitelist to mint new KVT tokens and transfer them to the KonverterMinter contract, and then lock them in the locker contract.

The KonverterMinter contract also includes a function called "setLocker" which allows the owner of the contract to set the address of the locker contract. This function is restricted to only allow the owner of the KonverterMinter contract to execute it.

2.1.2. konverter-contract-klaytn-audit

MasterChefKonverter.sol

This contract includes references to the "KonverterToken" contract and the "IKonverterVote" interface.

The MasterChefKonverter contract appears to be a farming contract that allows users to deposit liquidity provider (LP) tokens, earn rewards in the form of KLAY tokens, and withdraw their KLAY tokens at any time. The contract includes functions for adding new LP tokens to the contract, depositing and withdrawing LP tokens and KLAY tokens, and updating the pool

information for each LP token. It also includes functions for setting and updating the emission rate for KLAY tokens, as well as functions for setting the percentage of rewards that go to the dev address and the weight of the vote for each user.

The contract also includes several events that can be used to track changes to the contract state, such as adding a new LP token, updating the pool information, and setting the dev address or dev percent.

FeeSharingPool.sol

This is the contract for a fee sharing pool. It has functionality for depositing and withdrawing a staking token, as well as distributing rewards in a rewards token to users. The contract also has a number of state variables that store information about the pool such as the total supply of the staking token, the addresses of the staking and rewards tokens, and various parameters such as the reward rate and the dev fee percentage.

The contract has a number of functions that allow users to interact with it, such as deposit and withdraw for staking and unstaking tokens, reward for claiming rewards, and updateRewardsDuration and updateMultiplierMaximizedDuration for modifying certain pool parameters. The contract also has a number of view functions for accessing information about the pool and its users, such as balanceOf and allocated.

The contract inherits from PausableUpgradeable, RewardsDistributionRecipientUpgradeable, and ReentrancyGuardUpgradeable, which provide pause and emergency stop functionality, as well as protections against reentrancy attacks and functions for upgrading the contract. The contract also uses the SafeERC20 library for performing safe token transfers.

MigratorKKN.sol

The MigratorKKN contract is a contract that allows users to migrate "Listed PoolToken of StableSwap" to "3Pool in Konverter". It does this by taking an old "Listed PoolToken" as input and transferring the equivalent value in tokens from the new "3Pool in Konverter" to the user.

The contract has a state variable called newPool which is the address of the new "3Pool in Konverter". It also has a state variable called newPoolToken which is the address of the token associated with the new "3Pool in Konverter".

The contract has a function called migrate which takes an old "Listed PoolToken" and an amount as input and migrates the equivalent value in tokens from the new "3Pool in Konverter" to the user. The contract checks if the old "Listed PoolToken" is a token for a normal pool or a meta pool, and handles the migration accordingly. If the old "Listed PoolToken" is for a normal pool, the contract calls the _migrate function to migrate the tokens. If the old "Listed PoolToken" is for a meta pool, the contract first removes the meta pool token and gets the

equivalent amount of base pool token, and then calls the _migrate function to migrate the base pool token.

The contract also has a function called setOldPool which allows the contract owner to set an old "Listed PoolToken" as a target for migration. The contract checks if the old "Listed PoolToken" is a token for a normal pool or a meta pool, and sets the appropriate variables to track the old pool and its associated token.

The contract has a function called setServiceOpenTime which allows the contract owner to set a time at which the migration service will be available. The contract checks if the current time is after the service open time before allowing the migration to proceed.

MigratorKSP.sol

The MigratorKSP contract is a contract that allows users to migrate from old liquidity pools to new liquidity pools. It does this by allowing users to transfer their old liquidity pool tokens to the MigratorKSP contract, and then the contract will migrate the liquidity from the old pool to the new pool.

The contract has a few state variables:

- oldFactory: The address of the old factory that created the old liquidity pools.
- factory: The address of the new factory that creates the new liquidity pools.
- WKLAY: The address of the WKLAY token.
- desiredLiquidity: The amount of liquidity that the user wants to migrate.
- serviceOpenTime: The time at which the migration service becomes available.

The contract has a single public function, migrate, that allows users to migrate their old liquidity pool tokens to the new liquidity pools. The function takes in the address of the old liquidity pool and the amount of liquidity that the user wants to migrate. It first checks that the migration service is open and that the old liquidity pool is a valid target for migration. It then gets the addresses of the tokens in the old liquidity pool and creates a new liquidity pool with these tokens if one does not already exist. The contract then transfers the desired amount of liquidity from the old pool to the new pool and sends the migrated liquidity pool tokens back to the user.

The contract also has a fallback function, receive, that allows users to transfer their old liquidity pool tokens to the contract. It checks that the sender is an old factory and then accepts the transfer.

The contract has several events that allow users to track the migration process:

- Recovered: Triggered when the contract recovers liquidity from the old pool.
- MigrateKSP: Triggered when the contract migrates liquidity from the old pool to the new pool.

MigratorKSPTo3Pool.sol

The contract MigratorKSPTo3Pool is a contract that allows users to migrate "Listed PoolToken of Konverter" to "3Pool in Klayswap". It does this by allowing users to call the migrate function, which takes an old liquidity pool token as an argument and a desired amount to migrate.

The contract first checks that the current time is after the serviceOpenTime and that the old liquidity pool token is a target pool of this contract. It then removes the desired amount of liquidity from the old pool, adds the liquidity to the new pool, and returns the new liquidity pool token to the user.

The contract also has a setServiceOpenTime function that allows the owner to set the serviceOpenTime variable, which determines the earliest time that users can migrate their tokens. It also has a setTargetLps function that allows the owner to set the target liquidity pools for migration.

KonverterLockDrop.sol

The contract has a number of functions for users to interact with the farming program, such as deposit, withdraw, claimHarvestedKVT, and claimOldReward. It also has a migrate function that allows users to move their locked tokens to a different LP.

The contract maintains a list of LPs that are part of the farming program in the lockDropFarms set, and stores information about each LP in the lockDropFarmInfo array. It also maintains a mapping of users to the LPs they have locked tokens in, in the userLockDropFarms mapping.

The contract has an initialize function that is called to set the address of the KVT token, and a setKvtBuilder function that allows the contract owner to set the address of a contract for building the KVT token. The contract also has an event for when a user migrates their locked tokens to a different LP, and several other events for tracking deposits, withdrawals, and claims of rewards.

KonverterLockDropFarm.sol

The KonverterLockDropFarm contract is an upgradeable contract that allows users to deposit a token (called "oldLpToken") and receive a reward in the form of another token (called "oldRewardToken") based on the duration of the deposit. The contract is designed to be used in two phases: an "event period" where users can deposit oldLpToken and receive oldRewardToken, and a "reward distribution period" where users can claim their earned oldRewardToken. The KonverterLockDropFarm contract also has a function called migrate that allows users to convert their oldLpToken into a new token (called "newLpToken") after the event period has ended.

KonverterLockDropFarmKKN.sol

The KonverterLockDropFarmKKN contract is a variant of the KonverterLockDropFarm contract that is specifically designed to work with a token called "KKN" as the oldRewardToken. The KonverterLockDropFarmKKN contract has an additional function called deposit that allows users to deposit oldLpToken and receive KKN as a reward. This function also calls the stake function of the IMasterChefKKN contract to stake the deposited oldLpToken and claim any reward that is available.

KonverterLockDropFarmKSP.sol

The KonverterLockDropFarmKSP contract is a variant of the KonverterLockDropFarm contract that is specifically designed to work with a token called "KSP" as the oldRewardToken. The KonverterLockDropFarmKSP contract has an additional function called deposit that allows users to deposit oldLpToken and receive KSP as a reward. This function transfers oldLpToken to the KonverterLockDropFarmKSP contract and updates the reward balance for the depositing user based on the total supply of oldLpToken that has been deposited.

KonverterBuilder.sol

The KonverterBuilder contract is a contract that allows users to deposit KVT (Korean Veth) or Klaytn (KLAY) tokens to receive LP (liquidity provider) tokens for a specific pair. The contract also allows users to withdraw LP tokens, claim farmed KVT tokens, and claim event KVT tokens.

The contract has several state variables, including the addresses of the KVT, KLAY, and KVT-KLAY LP tokens, the router contract, the chef contract, and the lockDrop contract. It also has variables for the start and end dates of the event and the launch date of the contract.

There are also several mapping variables, including balanceKvt, which tracks the KVT balance of each user, balanceKlay, which tracks the KLAY balance of each user, and userLpPaid, which tracks the amount of LP tokens paid out to each user.

The contract has several functions, including depositKlay, which allows users to deposit KLAY tokens and receive LP tokens, withdrawLp, which allows users to withdraw LP tokens, claimFarmedKVT, which allows users to claim farmed KVT tokens, and claimEventKVT, which allows users to claim event KVT tokens. It also has a recover function that allows the contract owner to recover any accidentally sent tokens.

The contract also has several view functions, including shareOf, which returns the percentage of LP tokens a user holds before withdrawal, kvtShareOf, which returns the percentage of KVT tokens a user holds, and klayShareOf, which returns the percentage of KLAY tokens a user holds. It also has a withdrawableLpOf function, which returns the amount of LP tokens a user can withdraw.

SwapRouter.sol

This code is for a contract called SwapRouter in the Solidity programming language for the Klaytn platform. It appears to be a contract for exchanging tokens on the Klaytn blockchain.

The contract has a getAmountOut function that takes in an amountIn of a token, a path of tokens to be exchanged through, and an optional array of breakIndexes indicating where to switch between using the routerV2 contract and the stableSwapHelper contract to perform the exchanges. The function returns the estimated amountOut of the final token in the path.

The contract also has a swapExactTokensForTokens function that allows a user to exchange an amountIn of a token for an amountOut of another token, with the exchanges occurring according to the provided path and breakIndexes. The function takes in a deadline and to address, and will only execute the swap if the transaction is successful by the deadline.

The contract also has an initialize function that is called to set the routerV2 and stableSwapHelper addresses, and a receive function that allows it to receive token payments. It also imports and utilizes several other contracts, including OwnableUpgradeable, ReentrancyGuardUpgradeable, SafeERC20, IKonverterRouter02, IStableSwap, and IStableSwapHelper.

StableSwapHelper.sol

The StableSwapHelper contract is a utility contract that provides functions for interacting with stablecoin swap pools and meta pools. It utilizes the IStableSwapHelper, OwnableUpgradeable, and ReentrancyGuardUpgradeable contracts and the SafeERC20 and KonverterUtils libraries. The contract maintains a mapping of addresses to booleans that denotes whether an address is a base pool and a mapping of addresses to mappings of addresses to addresses that denotes the stablecoin swap pool for a given pair of token addresses. The contract has a number of view functions that allow users to retrieve information about stablecoin swap pools and meta pools, including the stablePoolOf function which returns the address of the stablecoin swap pool for a given pair of token addresses and the getAmountsOut function which returns the amounts of tokens at each point in a given path. The contract also has functions for calculating various quantities related to stablecoin swap pools and meta pools, including the calcTokenAmount function which calculates the token amount for a given pool based on given amounts of underlying tokens and the calcWithdrawOneCoin function which calculates the amount of a single underlying token that can be withdrawn from a given pool for a given amount of pool tokens.

KonverterFactory.sol

The KonverterFactory contract is a factory contract that allows users to create and manage KonverterPair contracts. A KonverterPair contract is a type of ERC20 token that is backed by

two other ERC20 tokens in a 1:1 ratio. This means that if you have 1 KonverterPair token, you can exchange it for 1 of the underlying ERC20 tokens at any time.

The KonverterFactory contract has a number of functions that allow users to manage the KonverterPair contracts. Users can create new KonverterPair contracts by calling the createPair function and specifying the two underlying ERC20 tokens. They can also set the fee that is charged for creating a new KonverterPair contract, set the address that will receive the fees, and set a migrator contract that will be able to migrate liquidity between KonverterPair contracts.

The KonverterFactory contract also has a number of functions that allow users to manage the KonverterPair contracts. Users can set the flash on or off by calling the setFlashOn function, set the flash fee by calling the setFlashFee function, and set the create fee by calling the setCreateFee function.

The KonverterFactory contract also has a number of functions that allow users to retrieve information about the KonverterPair contracts. Users can get the length of the allPairs array by calling the allPairsLength function, get the code hash of the KonverterPair contract by calling the pairCodeHash function, and get the address of a KonverterPair contract by calling the getPair function and specifying the underlying ERC20 tokens.

KonverterPair.sol

The KonverterPair contract is a smart contract that allows users to trade between two ERC20 tokens. It is part of a Konverter system, which includes the KonverterFactory contract that deploys new KonverterPair contracts, the KonverterERC20 contract that provides ERC20 functionality to the KonverterPair contracts, and the IMigrator interface that allows for migrating liquidity between KonverterPair contracts. The KonverterPair contract has functions for adding and removing liquidity, as well as for swapping between the two ERC20 tokens. It also has a lock modifier that can be applied to certain functions to prevent them from being executed if the contract is in a locked state. The contract keeps track of the reserves of the two ERC20 tokens, as well as price accumulators that are updated on the first call per block. It also includes various events for logging mint, burn, and swap actions on the contract.

KonverterVM.sol

The KonverterVM contract appears to be a contract that allows users to deposit and withdraw a token (KVT) and to use their KVT balance as a voting power in another contract (IKonverterVote). It also appears to have a system for calculating "accrued vote" and "remaining votes" based on the user's KVT balance, the amount of time that has passed since their last balance update, and a maximum period of time. The contract has a function to refund a user's accrued vote back to their KVT balance if they withdraw their KVT from the contract. It also has a function to calculate the estimated accrued vote that a user would receive if they

deposited a certain amount of KVT into the contract. The contract has an owner and an initialize function that sets the KVT token address. It also has a function to set the address of the IKonverterVote contract.

KonverterVote.sol

KonverterVote is a Solidity contract that allows users to vote for "LP" (liquidity provider) contracts that have been added to a vote list. Users can vote by calling the vote function and passing in the address of the LP contract they want to vote for and the amount they want to vote. The recallVote function allows users to recall their votes and the addVoteList and removeVoteList functions allow an owner of the contract to add or remove LP contracts from the vote list. The getVoteList and getDeprecatedList functions return the addresses of the LP contracts that are currently on the vote list or deprecated list, respectively. The contract also has a paused state that can be toggled on or off with the setPaused function, which will prevent users from voting if it is set to true. The userVoted function allows users to check how many votes they have cast for a specific LP contract and the userMileageUsed function allows users to check how many votes they have cast in total. The totalVote function returns the total number of votes that have been cast by all users. The lpToVote function returns the total number of votes that have been cast for a specific LP contract.

Zap.sol

The contract has a number of functions for estimating the amount of tokens that can be received when exchanging tokens or KLAY (the native cryptocurrency of Klaytn) with an LP. For example, the estimateZapTokenToPair function takes in an amount of a token from, and estimates the amount of two other tokens (amountA and amountB) that can be received by exchanging the from token with an LP identified by the to address. The function can handle the case where the from token is one of the tokens in the LP, or where the from token is not in the LP and must be first exchanged for another token that is in the LP.

The estimateZapKlayToPair function performs a similar calculation, but takes in an amount of KLAY and estimates the amount of the two tokens in the LP that can be received in exchange.

The contract also has a receive function that allows it to receive KLAY payments. It also imports and utilizes several other contracts, including Math, IKonverterPair, and ZapHelper.

ZapBasePool.sol

The ZapBasePool contract is a smart contract that allows users to convert ERC20 tokens to another ERC20 token which is managed by a liquidity pool. The contract uses a ROUTER contract to find the best path to convert the ERC20 token to the liquidity pool token.

The initialize function is called to set up the contract and it takes three arguments: _router, _zapRoute, and _basePool. The _router argument is the address of the contract that will be used

to find the best path to convert the ERC20 token to the liquidity pool token. The _zapRoute argument is the address of a contract that determines the fee that will be charged for the conversion. The _basePool argument is the address of the contract that manages the liquidity pool and the liquidity pool token.

The estimateZapToBasePoolToken function allows users to estimate the amount of liquidity pool token they will receive when they convert a certain amount of an ERC20 token to the liquidity pool token. The function takes two arguments: from, the address of the ERC20 token to be converted, and amountIn, the amount of the ERC20 token to be converted. The function returns an array of the estimated amounts of each underlying coin in the liquidity pool and the estimated amount of liquidity pool token that the user will receive.

The zapTokenToBasePoolToken function allows users to actually convert an ERC20 token to the liquidity pool token. The function takes three arguments: from, the address of the ERC20 token to be converted, amountIn, the amount of the ERC20 token to be converted, and amountOutMin, the minimum amount of liquidity pool token that the user is willing to accept. The function first finds the best path to convert the ERC20 token to the liquidity pool token using the ROUTER contract, then converts the ERC20 token to the liquidity pool token using the addLiquidity function of the basePool contract. The function also charges a fee for the conversion using the zapRoute contract.

ZapMetaPool.sol

The ZapMetaPool contract is a smart contract that allows users to exchange various ERC20 tokens for a token specific to a "meta pool", which is a group of liquidity pools that share a common denominator. The contract uses the ZapHelper contract which provides some shared functionality such as estimating the amount of tokens that will be returned for a given input amount and calculating the best path for exchanging tokens.

The contract has several state variables that store information about the meta pool and its associated base pool, as well as lists of the different ERC20 tokens that are supported by the pools. It also has a receive function that allows the contract to receive ERC20 tokens.

The contract provides a view function called estimateZapToMetaPoolToken that allows users to estimate the amount of meta pool tokens that they will receive for a given amount of an ERC20 token. The function first checks if the input token is already one of the coins supported by the meta pool. If it is, the function returns zero meta pool tokens. Otherwise, it calculates the best path for exchanging the input token for a coin supported by the meta pool or the base pool, estimates the amount of that coin that will be received, and calculates the corresponding amount of meta pool tokens. The function returns the estimated amounts of underlying coins and the estimated amount of meta pool tokens.

2.2. Findings

During the audit process, the audit team found some vulnerability issues in the given version of Konverter.

#	Issue	Severity	Status
1	Incorrect order of arithmetic operations	CRITICAL	FIXED
2	Incorrect condition checking	CRITICAL	FIXED
3	The `_userVoted` is incorrectly updated in the `vote` function	CRITICAL	FIXED
4	`feeToken` is incorrectly approved for the contract deployer	MEDIUM	FIXED
5	LP token that follows the ERC777 standard can cause re-entrancy problems	LOW	FIXED
6	Missing front-running protection for the `approve` function	LOW	FIXED
7	Wrong timestamp checking lead to be inconsistent code	LOW	FIXED

Table 3. The discovered vulnerabilities

2.2.1. Incorrect order of arithmetic operations - CRITICAL

Affected files:

- swap/cp-amm/KonverterPair.sol

In the `swap` function of the `KonverterPair` contract, the calculation of `amount0Out` and `amount1Out` are incorrect due to the use of `*=` operators. In this context, multiplications must be performed before divisions.

```
function swap(
    uint256 amount0Out,
    uint256 amount1Out,
    address to,
    bytes calldata data
) external lock {
    // ...
    {
        // scope for _token{0,1}, avoids stack too deep errors
        address _token0 = token0;
        address _token1 = token1;
        require(to != _token0 && to != _token1, "KVT: INVALID_TO");
        if (amount0Out > 0) _safeTransfer(_token0, to, amount0Out); // optimistically
```

```
transfer tokens
        if (amount1Out > 0) _safeTransfer(_token1, to, amount1Out); // optimistically
transfer tokens
        if (IKonverterFactory(factory).flashOn() && data.length > 0) {
            if (amount0Out > 0) {
                _safeTransfer(_token0, IKonverterFactory(factory).feeTo(), amount0Out *
IKonverterFactory(factory).flashFee() / 10000);
                amount0Out *= (10000 + IKonverterFactory(factory).flashFee()) / 10000; //
INCORRECT
            }
            if (amount1Out > 0) {
                _safeTransfer(_token1, IKonverterFactory(factory).feeTo(), amount1Out *
IKonverterFactory(factory).flashFee() / 10000);
                amount1Out *= (10000 + IKonverterFactory(factory).flashFee()) / 10000;  //
INCORRECT
            }
            IKonverterCallee(to).konverterCall(msg.sender, amount0Out, amount1Out, data);
        }
        balance0 = IERC20(_token0).balanceOf(address(this));
        balance1 = IERC20(_token1).balanceOf(address(this));
    }
    // ...
}
```

RECOMMENDATION

The code above can be fixed as below.

```
function swap(
    uint256 amount0Out,
    uint256 amount1Out,
    address to,
    bytes calldata data
) external lock {
    // ...
    {
        // scope for _token{0,1}, avoids stack too deep errors
        address _token0 = token0;
        address _token1 = token1;
        require(to != _token0 && to != _token1, "KVT: INVALID_TO");
        if (amount0Out > 0) _safeTransfer(_token0, to, amount0Out); // optimistically
transfer tokens
        if (amount1Out > 0) _safeTransfer(_token1, to, amount1Out); // optimistically
transfer tokens
        if (IKonverterFactory(factory).flashOn() && data.length > 0) {
            if (amount0Out > 0) {
                _safeTransfer(_token0, IKonverterFactory(factory).feeTo(), amount0Out *
IKonverterFactory(factory).flashFee() / 10000);
                amount0Out = amount0Out * (10000 + IKonverterFactory(factory).flashFee()) /
10000; // FIXED
            }
```

```
            if (amount1Out > 0) {
                _safeTransfer(_token1, IKonverterFactory(factory).feeTo(), amount1Out *
IKonverterFactory(factory).flashFee() / 10000);
                amount1Out = amount1Out * (10000 + IKonverterFactory(factory).flashFee()) /
10000;  // FIXED
            }
            IKonverterCallee(to).konverterCall(msg.sender, amount0Out, amount1Out, data);
        }
        balance0 = IERC20(_token0).balanceOf(address(this));
        balance1 = IERC20(_token1).balanceOf(address(this));
    }
    // ...
}
```

UPDATES

- *Dec 23, 2022*: This issue has been acknowledged and fixed by the Konverter team.

2.2.2. Incorrect condition checking - CRITICAL

Affected files:

- migration/MigratorKSP.sol

In the `migrate` function of the `MigratorKSP` contract, there is an incorrect checking condition between two variables (`tokenA` and `tokenB`) that leads to incorrect program execution flow.

```
if (tokenA == address(0)) {
    tokenA = WKLAY;
} else if (tokenA == address(0)) { // INCORRECT
    tokenB = WKLAY;
}
```

RECOMMENDATION

The above code can be fixed as below.

```
if (tokenA == address(0)) {
    tokenA = WKLAY;
} else if (tokenB == address(0)) { // FIXED
    tokenB = WKLAY;
}
```

UPDATES

- *Dec 23, 2022*: This issue has been acknowledged and fixed by the Konverter team.

2.2.3. The `_userVoted` is incorrectly updated in the `vote` function - CRITICAL

Affected files:

- vote/KonverterVote.sol

In the `vote` function of the `KonverterVote` contract, the state variable `_userVoted[user][lp]` is incorrectly updated. This variable must be increased by the `amount` value, in this case, the votes of that user for the specified pool will be lower than the actual amount if the `vote` function is called multiple times.

```
function vote(address user, address lp, uint amount) external override onlyVmContract {
    require(!_paused, "KonverterVote: paused");
    require(_voteList.contains(lp), "KonverterVote: LP does not exist");
    uint pid = IMasterChefKonverter(masterChef).pidOf(lp);

    _totalVote = _totalVote + amount;
    _lpToVote[lp] = _lpToVote[lp] + amount;
    _userVoted[user][lp] = amount; // INCORRECT
    _userMileageUsed[user] = _userMileageUsed[user] + amount;

    IMasterChefKonverter(masterChef).updateBoostRatioBehalf(pid, user);

    emit Vote(user, lp, amount);
}
```

RECOMMENDATION

The code above can be fixed as below.

```
function vote(address user, address lp, uint amount) external override onlyVmContract {
    require(!_paused, "KonverterVote: paused");
    require(_voteList.contains(lp), "KonverterVote: LP does not exist");
    uint pid = IMasterChefKonverter(masterChef).pidOf(lp);

    _totalVote = _totalVote + amount;
    _lpToVote[lp] = _lpToVote[lp] + amount;
    _userVoted[user][lp] += amount; // FIXED
    _userMileageUsed[user] = _userMileageUsed[user] + amount;

    IMasterChefKonverter(masterChef).updateBoostRatioBehalf(pid, user);

    emit Vote(user, lp, amount);
}
```

UPDATES

- *Dec 23, 2022*: This issue has been acknowledged and fixed by the Konverter team.

2.2.4. `feeToken` is incorrectly approved for the contract deployer - MEDIUM

Affected files:

- feeSharing/MoneyMakerKonverter.sol

In the `initialize` function of the `MoneyMakerKonverter` contract, the `feeToken` is approved for the `feeSharingPool`. However, the `feeSharingPool` variable is not initialized at this time, so the `feeToken` is being approved for the contract deployer which is incorrect. Moreover, we don't see any function in the `FeeSharingPool` contract that need the `feeToken` approval from the `MoneyMakerKonverter` contract. So, consider removing the token approval statement if it's not necessary.

```
function initialize(
    address _factory,
    address _feeToken,
    address _wklay,
    address _stableSwapHelper
) external initializer {
    require(_factory != address(0), "MoneyMakerKonverter: factory can't be address(0)");
    require(_feeToken != address(0), "MoneyMakerKonverter: token can't be address(0)");
    require(_wklay != address(0), "MoneyMakerKonverter: wklay can't be address(0)");
    require(_stableSwapHelper != address(0), "MoneyMakerKonverter: _stableSwapHelper can't
be address(0)");
    __Ownable_init();
    FACTORY = _factory;
    feeToken = _feeToken;
    WKLAY = _wklay;
    bridges[WKLAY] = _feeToken;
    stableSwapHelper = _stableSwapHelper;
    feeSharingPool = _msgSender(); // @dev call setFeeSharingPool after feeSharingPool
deployed
    devCut = 3333;
    devAddr = _msgSender();
    authList.add(_msgSender());
    IERC20(feeToken).approve(feeSharingPool, type(uint).max); // INCORRECT
}
```

RECOMMENDATION

Remove the approve-call statement above.

UPDATES

- *Dec 23, 2022*: This issue has been acknowledged and fixed by the Konverter team.

2.2.5. LP token that follows the ERC777 standard can cause reentrancy problems - LOW

Affected files:

- farm/MasterChefKonverter.sol

The `MasterChefKonverter` contract allows the admin to add pools for staking `lpToken` and get the `konverter` token as the reward. However, if the `lpToken` follows the ERC777 standard (*https://docs.openzeppelin.com/contracts/2.x/api/token/erc777#IERC777Recipient*) that supports `tokensToSend` and `tokensReceived` hooks, the attacker can deploy a smart contract with these hooks as a receiver to perform a reentrancy attack that calls back to this contract in the same transaction. The attack can happen since the `lpToken` is transferred back to the user before the contract state is completely updated.

```
function withdraw(uint pid, uint amountOut) public override {
    PoolInfo storage pool = poolInfo[pid];
    UserInfo storage user = userInfo[pid][msg.sender];
    require(user.amount >= amountOut, "MasterChef: bad amount");

    updatePool(pid);

    // Harvest KVT
    uint pending = ((user.amount * user.boostRatio * pool.accKonverterPerShare) / 1e12 /
10000) - user.rewardDebt;
    safeKonverterTransfer(msg.sender, pending);
    emit Harvest(msg.sender, pid, pending);

    pool.totalBoostedAmount = pool.totalBoostedAmount - (user.amount * user.boostRatio /
10000);
    user.amount = user.amount - amountOut;

    pool.totalAmount = pool.totalAmount - amountOut;
    IERC20(pool.lpToken).safeTransfer(msg.sender, amountOut); // VULNERABLE TO REENTRANCY
ATTACKS

    if (user.amount < 1000) {
        // clean up dust and remove farmList
        uint dust = user.amount;
        user.amount = 0;
        pool.totalAmount = pool.totalAmount - dust;
        IERC20(pool.lpToken).safeTransfer(msg.sender, dust);
        userFarmList[msg.sender].remove(pid);
    }
    user.boostRatio = pendingBoostRatioOf(pid, msg.sender);
    user.rewardDebt = (user.amount * user.boostRatio * pool.accKonverterPerShare / 1e12) /
10000;
    pool.totalBoostedAmount = pool.totalBoostedAmount + (user.amount * user.boostRatio /
10000);
    emit Withdraw(msg.sender, pid, amountOut);
}
```

RECOMMENDATION

All the external calls that transfer the token back to user should be put at the end of the function. With this pattern, we can make sure that the contract state has been completely updated before calling to external contracts. This pattern is called as "checks effects interactions".

UPDATES

- *Jan 09, 2023*: This issue has been fixed following our recommendations in commit `2e90998144a6edacbddf76128c46cbf597796609`.

2.2.6. Missing front-running protection for the `approve` function - LOW

Affected files:

- tokens/KonverterERC20.sol
- tokens/StableSwapPoolToken.sol

The `approve` functions for both `KonverterERC20` and `StableSwapPoolToken` do not have any mechanism to protect the logic against front-running attacks. For more information about front-running attack related to the `approve` function, refer to the following document.

https://docs.google.com/document/d/1YLPtQxZu1UAvO9cZ1O2RPXBbT0mooh4DYKjA_jp-RLM

RECOMMENDATION

Adding the `increaseAllowance` and `decreaseAllowance` functions to the logic of the ERC20 token contracts or modifying the logic of the `approve` function so that it only allows changing the allowance value from/to zero.

UPDATES

- *Dec 23, 2022*: The above issue has been fixed by adding the `increaseAllowance` and `decreaseAllowance` functions to mentioned ERC20 contracts.

2.2.7. Wrong timestamp checking lead to be inconsistent code - LOW

Affected files:

- preLaunch/KonverterLockDropFarm.sol

While project defines period 1 less than or equal to (<=) `phase1EndTime` and owner can only `migrate` when timestamp greater than (>) `phase2EndTime`, but `isPhase2EventPeriod` was defined greater than or equal to (>=) `phase1EndTime` and less than `phase2EndTime`. And so, there is an intersection point of phase 1 and phase 2. Besides, there is a gap between phase 2 and migrated time.

```
function isPhase2EventPeriod() public override view returns (bool) {
    return (block.timestamp >= phase1EndTime && block.timestamp < phase2EndTime);
//INCORRECT
}
```

RECOMMENDATION

The code above can be fixed as below.

```
function isPhase2EventPeriod() public override view returns (bool) {
    return (block.timestamp > phase1EndTime && block.timestamp <= phase2EndTime); //FIXED
}
```

UPDATES

- *Jan 09, 2023*: This issue has been acknowledged and fixed by the Konverter team in commit `2e90998144a6edacbddf76128c46cbf597796609`.

3. VERSION HISTORY

Version	Date	Status/Change	Created by
1.0	*Dec 22, 2022*	Private Report	Verichains Lab
1.1	*Jan 09, 2023*	Public Report	Verichains Lab

Table 4. Report versions history