Smart Contract Security Audit

Bamboo DeFi

2021-01-29

1. Introduction

BAMBOO DEFI is designed to be the global reference platform used to exchange, save and "cultivate" cryptocurrencies with the best possible ratio without putting in risk the stability of the project, taking advantage of well-tested and audited protocols.

The BambooDeFi project primarily derives its functionalities from Uniswap V2 and implement the SushiSwap migration, but additionally integrating functionalities, such as an incentive program based on multipliers in Yield Farming + Yield Farming with blocking Staking. Therefore, the project wants to analyze the risks derived from its implementation.

As requested by **Bamboo DeFi** and as part of the vulnerability review and management process, **Red4Sec** has been asked to perform a security code audit in order **to evaluate the security of the Bamboo DeFi Smart Contract source code.**

2. Disclaimer

This document only represents the results of the code audit conducted by Red4Sec Cybersecurity and should not be used in any way to make investment decisions or as investment advice on a project.

Likewise, the report should not be considered neither "endorsement" nor "disapproval" of the guarantee of the correct business model of the analyzed project.

3. Scope

The detailed audit below has been based on the following commit *bc4284f150602744ad9884fc6cc704f26affd7a1* and subsequent revisions up to commit *fd1a7259f7bc60566dcff41582935962a5701630*.

- https://github.com/bamboo-defi/bamboodefi

The scope of this evaluation includes the following components:

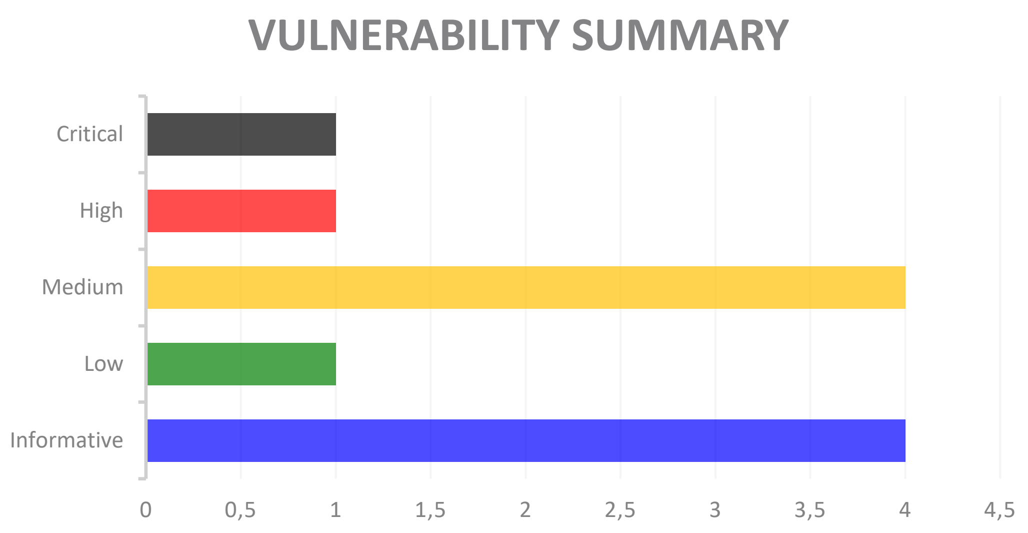
- BambooField
- BambooFarmer
- ZooKeeper
- Raindrop
- BBYP
- Factory
- Router

4. Conclusions

To this date, 29th of January 2021, the general conclusion resulting from the conducted audit, is that the **BambooDeFi Smart Contracts is secure** and does not present any critical-high known vulnerabilities that could compromise the security of the users and their information, although Red4Sec has found a few potential improvements.

- During the security audit, a total of 11 vulnerabilities and miscellaneous issues were detected some of them were critical-high risk. Keep in mind that some of these vulnerabilities do not pose any risk by themselves and have been classified as informative. Consequently, the developers started the mitigation process immediately after being informed.
- Bamboo team maintains some centralized parts that imply trust in the project until these permissions are transferred to future governance. Note that *Migrate*, *Timelock* and *Governance* contracts still have pending development.
- A few low impact issues were detected and classified only as informative, but they will continue to help Bamboo improve the security and quality of its developments.

All these vulnerabilities have been classified in the following levels of risk according to the impact level defined by CVSS v3 (*Common Vulnerability Scoring System*) by the National Institute of Standards and Technology (*NIST*):



Below we have a complete list of the vulnerabilities detected by Red4Sec, presented and summarized in a way that can be used for risk management and mitigation.

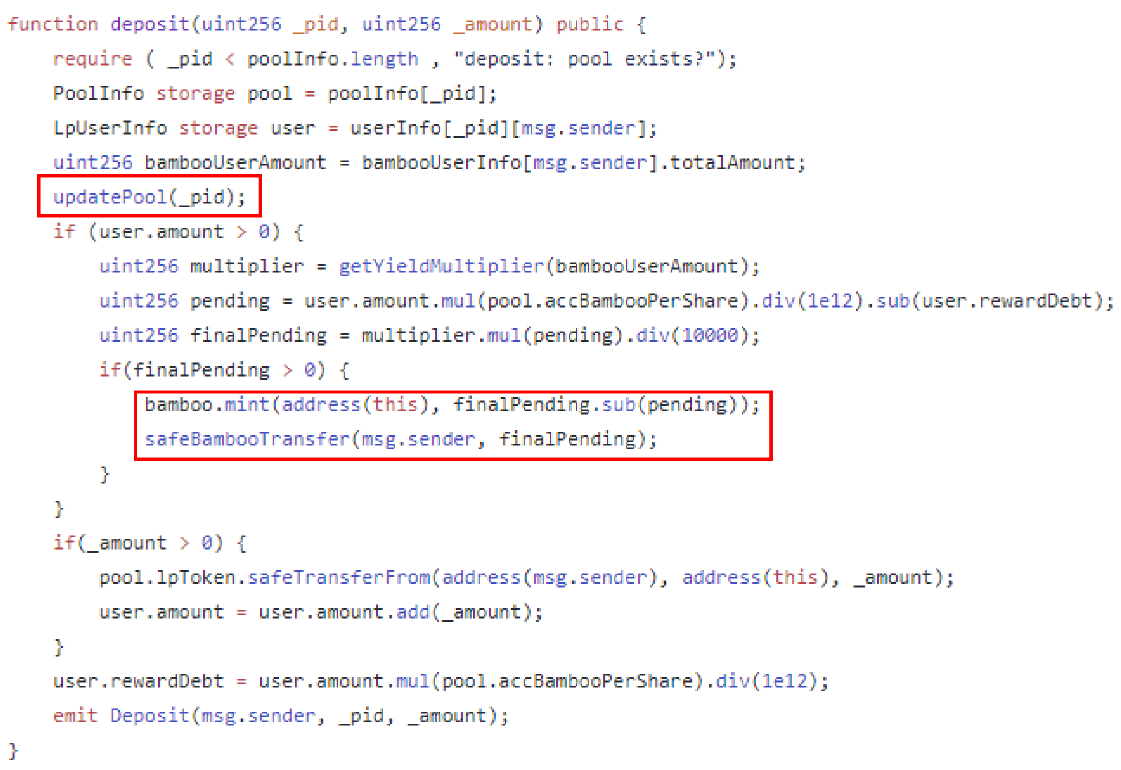
Table of vulnerabilities

Id.	Vulnerability	Risk	State
BD001	Unlimited Bamboo Minting	Critical	Fixed
BD002	Unbounded Loop in Withdrawal methods	High	Fixed
BD003	Unsafe ERC20 transfers	Medium	Fixed
BD004	Contracts Management Risks	Medium	Partially Fixed
BD005	Lack of Inputs Validation	Medium	Partially Fixed
BD006	Use of SafeMath	Medium	Fixed
BD007	Outdated Compiler Version	Informative	Fixed
BD008	Outdated Third-Party Libraries	Low	Fixed
BD009	GAS Optimization	Informative	Partially Fixed
BD010	Solidity Literals	Informative	Fixed
BD011	Code Style	Informative	Fixed

5. Issues and Recommendations

BD001. Unlimited Bamboo Minting

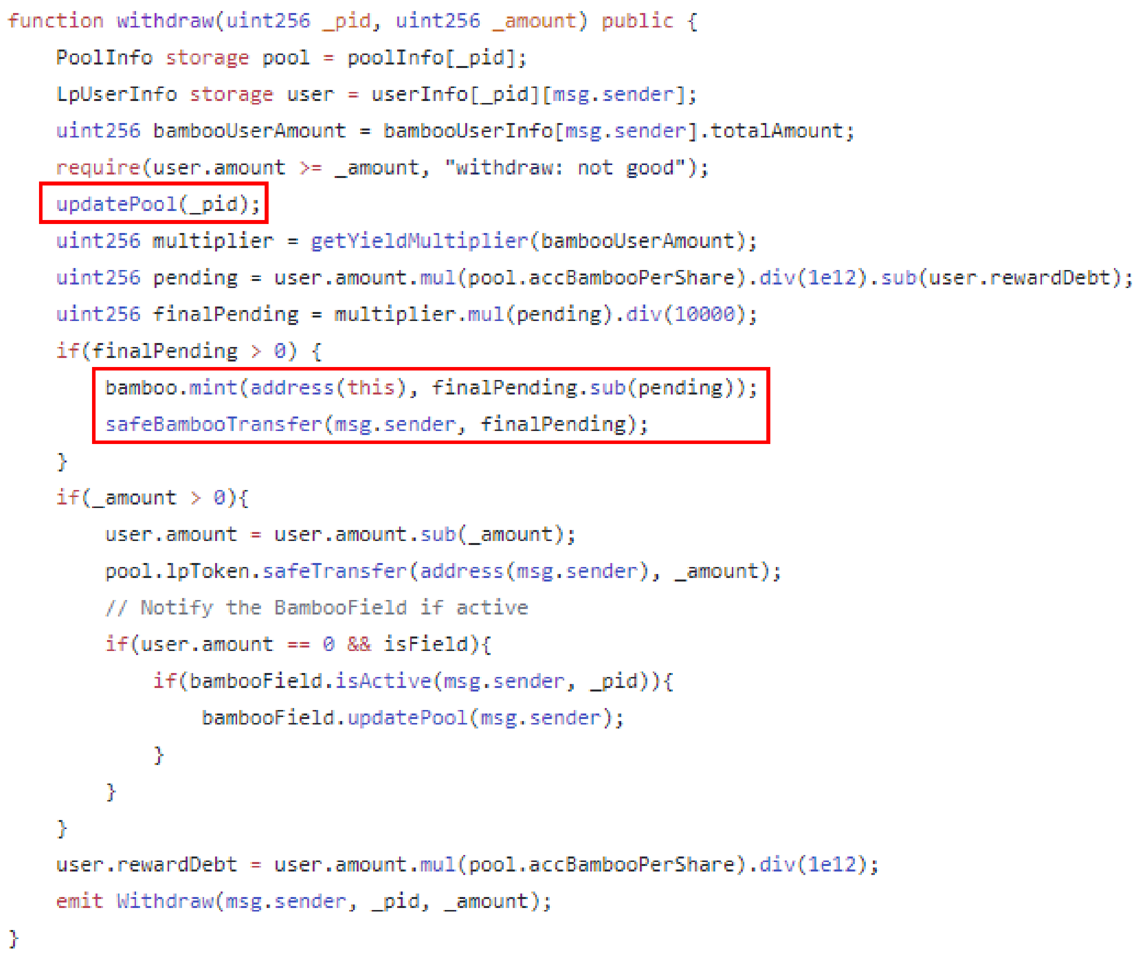
The **ZooKeeper** contract allows to mine Bamboo without limits. During the deposits of LP tokens to **Zookeeper**, Bamboo is generated as a reward to the liquidity provider, the problem is that the generated Bamboo is not time-dependent which allows to immediately withdraw liquidity, even in the same transaction, but maintaining the generated Bamboo.



```
function deposit(uint256 _pid, uint256 _amount) public {
    require ( _pid < poolInfo.length , "deposit: pool exists?");
    PoolInfo storage pool = poolInfo[_pid];
    LpUserInfo storage user = userInfo[_pid][msg.sender];
    uint256 bambooUserAmount = bambooUserInfo[msg.sender].totalAmount;
    updatePool(_pid);
    if (user.amount > 0) {
        uint256 multiplier = getYieldMultiplier(bambooUserAmount);
        uint256 pending = user.amount.mul(pool.accBambooPerShare).div(1e12).sub(user.rewardDebt);
        uint256 finalPending = multiplier.mul(pending).div(10000);
        if(finalPending > 0) {
            bamboo.mint(address(this), finalPending.sub(pending));
            safeBambooTransfer(msg.sender, finalPending);
        }
    }
    if(_amount > 0) {
        pool.lpToken.safeTransferFrom(address(msg.sender), address(this), _amount);
        user.amount = user.amount.add(_amount);
    }
    user.rewardDebt = user.amount.mul(pool.accBambooPerShare).div(1e12);
    emit Deposit(msg.sender, _pid, _amount);
}
```

The problem is because Bamboo tokens are mined twice; first in the *updatePool* function, which mines the tokens based on the last reward, and later on *deposit* the *safeBambooTransfer* function will exclusively repay the expected tokens.

The bamboo team, by including an extra mining in the *withdraw* function *(line 376)* manages to guarantee that the user will obtain Bamboo tokens even if their contribution is deposited and withdrawn in the same transaction.



```
function withdraw(uint256 _pid, uint256 _amount) public {
    PoolInfo storage pool = poolInfo[_pid];
    LpUserInfo storage user = userInfo[_pid][msg.sender];
    uint256 bambooUserAmount = bambooUserInfo[msg.sender].totalAmount;
    require(user.amount >= _amount, "withdraw: not good");
    updatePool(_pid);
    uint256 multiplier = getYieldMultiplier(bambooUserAmount);
    uint256 pending = user.amount.mul(pool.accBambooPerShare).div(1e12).sub(user.rewardDebt);
    uint256 finalPending = multiplier.mul(pending).div(10000);
    if(finalPending > 0) {
        bamboo.mint(address(this), finalPending.sub(pending));
        safeBambooTransfer(msg.sender, finalPending);
    }
    if(_amount > 0){
        user.amount = user.amount.sub(_amount);
        pool.lpToken.safeTransfer(address(msg.sender), _amount);
        // Notify the BambooField if active
        if(user.amount == 0 && isField){
            if(bambooField.isActive(msg.sender, _pid)){
                bambooField.updatePool(msg.sender);
            }
        }
    }
    user.rewardDebt = user.amount.mul(pool.accBambooPerShare).div(1e12);
    emit Withdraw(msg.sender, _pid, _amount);
}
```

This problem allows the users with plenty of liquidity, or simply through flash loans, to generate large volumes of Bamboo, without really contributing and without any risk.

Source reference

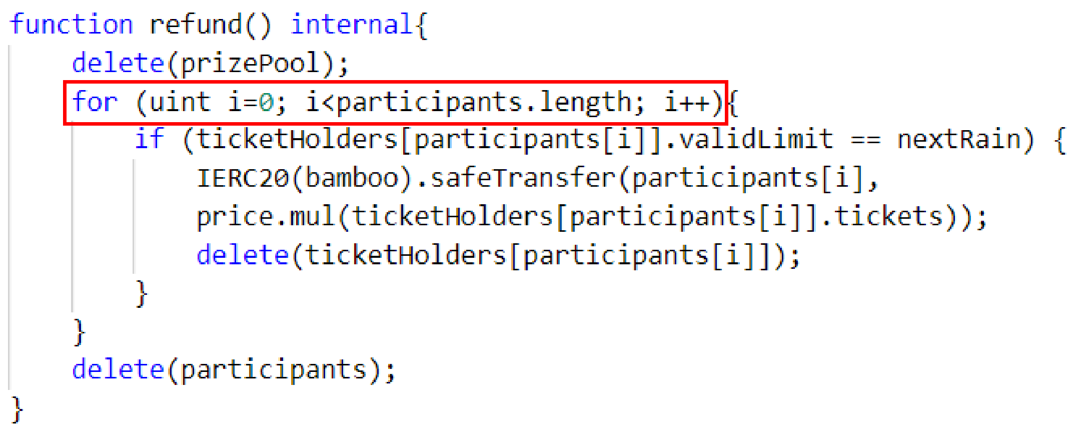
- ZooKeeper.sol: 305, 366

BD002. Unbounded Loop in Withdrawal methods

A few logics of the contract execute loops that can make too many iterations, which can trigger a denial of service (DoS) by GAS exhaustion because it iterates without any limit.

Loops without limits are considered a bad practice in the development of smart contracts, since they can trigger a denial of service (DoS) or overly expensive executions, this is the case affecting Bamboo.

A denial of service to these methods by a third party is possible, creating a large number of participants, winners, etc. this will allow to force that the cost of iterating all the inputs surpasses the maximum allowed GAS per block, which currently is of 12 million approximately.



```
function refund() internal{
    delete(prizePool);
    for (uint i=0; i<participants.length; i++){
        if (ticketHolders[participants[i]].validLimit == nextRain) {
            IERC20(bamboo).safeTransfer(participants[i],
            price.mul(ticketHolders[participants[i]].tickets));
            delete(ticketHolders[participants[i]]);
        }
    }
    delete(participants);
}
```

This Unbounded loop issue was found in the **ZooKeeper** and **RainDrop** contracts.

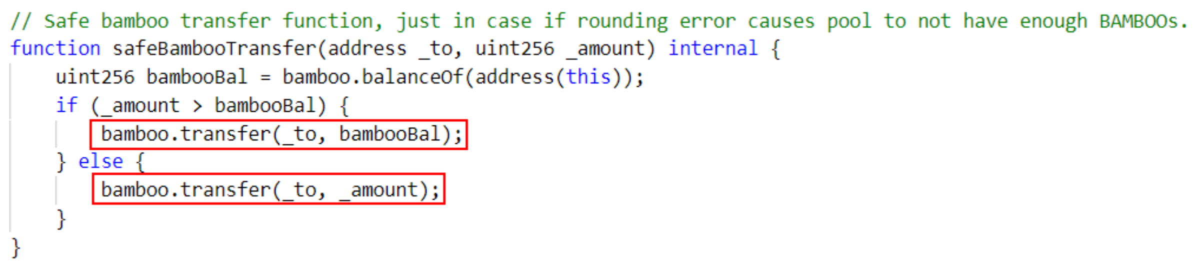
Code References

- Raindrop.sol: 134, 207
- ZooKeeper: 223, 285, 488

BD003. Unsafe ERC20 transfers

The definition of the *ERC-20*[1] standard states that the *transfer* and *transferFrom* methods must return a *boolean* value that determines whether the result was successful or not.

Throughout the audited contracts there are calls to the *transfer* methods that do not verify this result, leaving the open possibility that the execution is not as expected by the developer.



```
// Safe bamboo transfer function, just in case if rounding error causes pool to not have enough BAMBOOs.
function safeBambooTransfer(address _to, uint256 _amount) internal {
    uint256 bambooBal = bamboo.balanceOf(address(this));
    if (_amount > bambooBal) {
        bamboo.transfer(_to, bambooBal);
    } else {
        bamboo.transfer(_to, _amount);
    }
}
```

We recommend that you check the returned value using the *require* clause or unify these calls using the *safeTransfer* and *safeTransferFrom* wrappers of OpenZeppelin.

This problem can increase in the *buy* function of **BambooField**, since the only verification of the *_amount* argument is *transferFrom* which is executed at the end of the method, and if it returns '*false*', it would allow the user to mine arbitrary amounts of tokens.

Source references

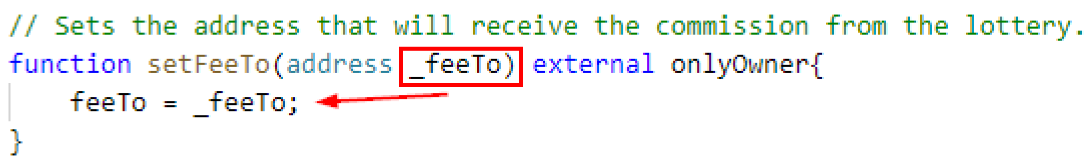
- BambooField.sol: 48, 77, 91, 107
- ZooKeeper.sol: 482, 484

[1] https://eips.ethereum.org/EIPS/eip-20#methods

BD004. Contracts Management Risks

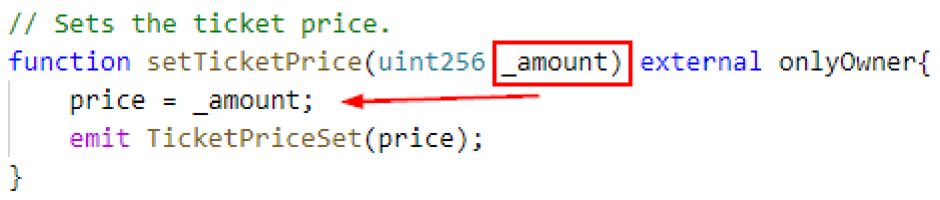
The logic design of the BambooDeFi contracts imply a few minor risks that should be reviewed and considered for their improvement.

- Since the owner is allowed to alter certain values arbitrarily in the lottery at any given time, such as the ticket price and the address of fee, it is advisable that the owner is not allowed to buy tickets in the **Raindrop** and **BBYP** contracts.
    - The address that receives the lottery fee may be altered by the owner through the *setFeeTo* method.



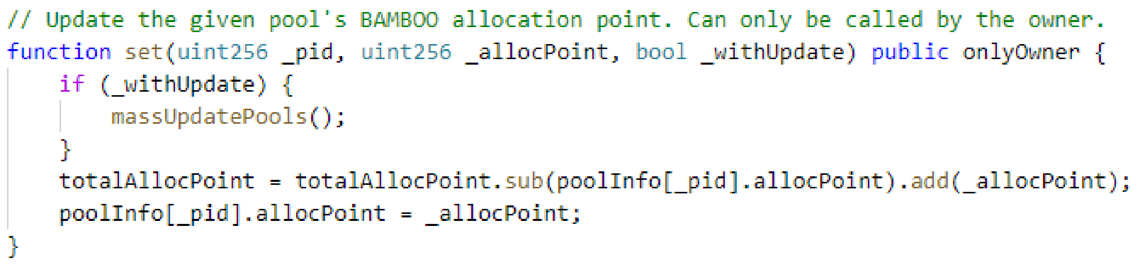
```
// Sets the address that will receive the commission from the lottery.
function setFeeTo(address _feeTo) external onlyOwner{
    feeTo = _feeTo;
}
```

    - In the following image, it can be observed that the ticket's price can be altered through the *setTicketPrice* method.

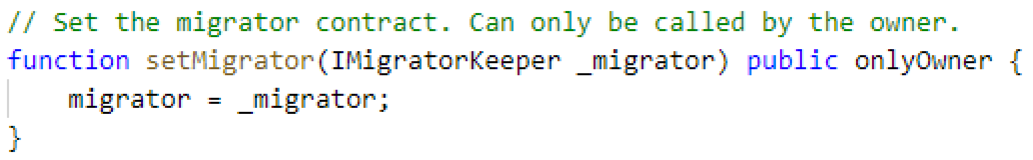


```
// Sets the ticket price.
function setTicketPrice(uint256 _amount) external onlyOwner{
    price = _amount;
    emit TicketPriceSet(price);
}
```

- The *setTicketPrice* of the **Raindrop** contract, allows to change the ticket's price at any given moment, this may cause a disparity between the cost and the refund made by the user, since the actual amount bought by the user is never stored.
- ZooKeeper's *migrate* method allows any user to migrate liquidity to new BambooDeFi pools, which initially is not a problem. However, it is recommended to add the *onlyOwner* modifier so that the project can control that the migration is carried out progressively, controlling possible derived issues and preventing possible front-runners, so that risk management is delegated exclusively to the person in charge of the project.
- There is no limitation that prevents the owner to call the *set* or *setMigrator* methods of the **ZooKeeper** contract at any given time, which would require the trust of a third party to avoid altering the contract's logic.

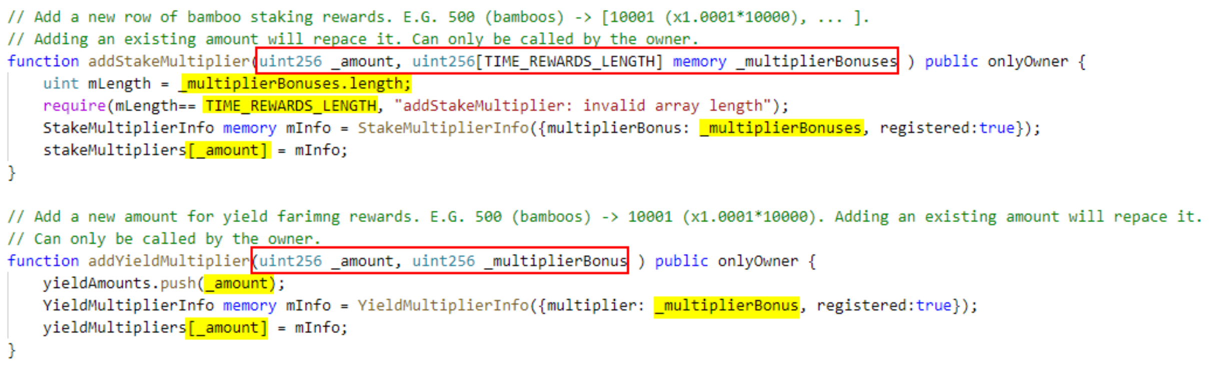


```
// Update the given pool's BAMBOO allocation point. Can only be called by the owner.
function set(uint256 _pid, uint256 _allocPoint, bool _withUpdate) public onlyOwner {
    if (_withUpdate) {
        massUpdatePools();
    }
    totalAllocPoint = totalAllocPoint.sub(poolInfo[_pid].allocPoint).add(_allocPoint);
    poolInfo[_pid].allocPoint = _allocPoint;
}
```



```
// Set the migrator contract. Can only be called by the owner.
function setMigrator(IMigratorKeeper _migrator) public onlyOwner {
    migrator = _migrator;
}
```

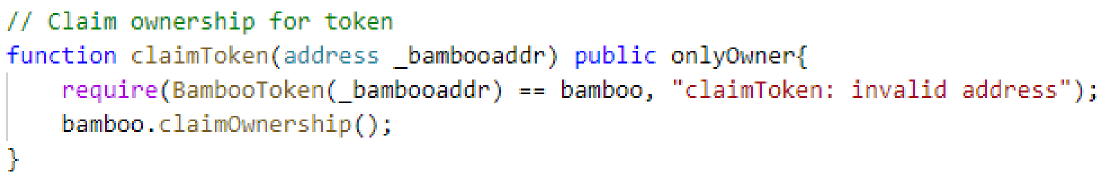
- The owner can alter the metrics through the *addStakeMultiplier* and *addYieldMultiplier* methods of the **ZooKeeper** contract, since the inputs and the entered quantities are never checked.



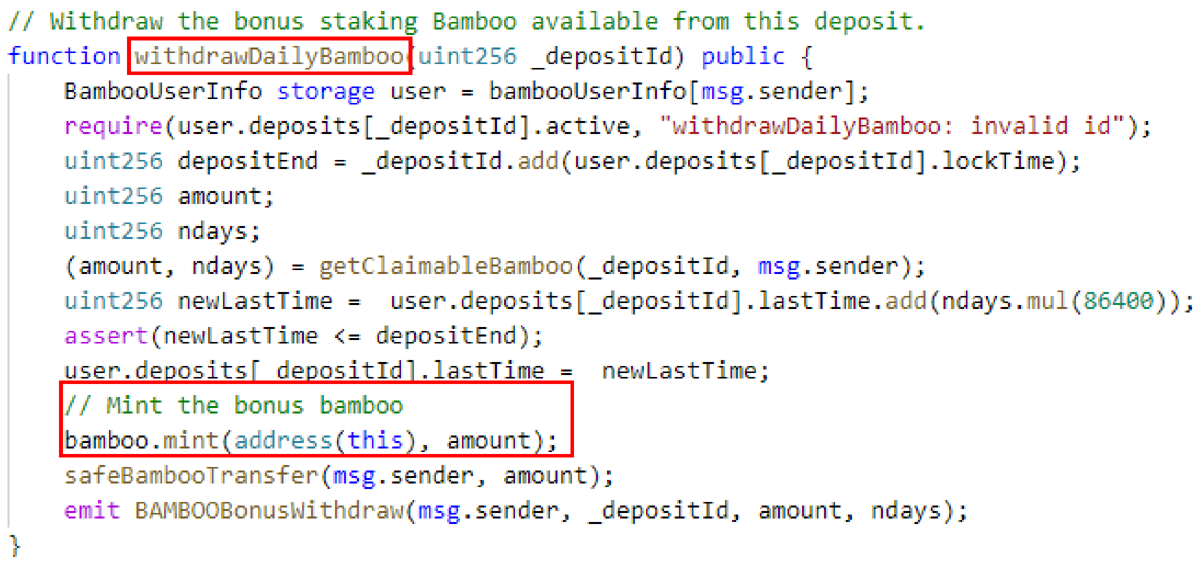
```
// Add a new row of bamboo staking rewards. E.G. 500 (bamboos) -> [10001 (x1.0001*10000), ... ].
// Adding an existing amount will repace it. Can only be called by the owner.
function addStakeMultiplier(uint256 _amount, uint256[TIME_REWARDS_LENGTH] memory _multiplierBonuses ) public onlyOwner {
    uint mLength = _multiplierBonuses.length;
    require(mLength== TIME_REWARDS_LENGTH, "addStakeMultiplier: invalid array length");
    StakeMultiplierInfo memory mInfo = StakeMultiplierInfo({multiplierBonus: _multiplierBonuses, registered:true});
    stakeMultipliers[_amount] = mInfo;
}

// Add a new amount for yield farimng rewards. E.G. 500 (bamboos) -> 10001 (x1.0001*10000). Adding an existing amount will repace it.
// Can only be called by the owner.
function addYieldMultiplier(uint256 _amount, uint256 _multiplierBonus ) public onlyOwner {
    yieldAmounts.push(_amount);
    YieldMultiplierInfo memory mInfo = YieldMultiplierInfo({multiplier: _multiplierBonus, registered:true});
    yieldMultipliers[_amount] = mInfo;
}
```

- It is important to mention that if the *claimOwnership* is not made through the *claimToken* method of the **ZooKeeper** contract, the mining made in the *withdrawDailyBamboo* method will not work. Besides, the *_bambooaddr* argument is not necessary for the logic of the function.



```
// Claim ownership for token
function claimToken(address _bambooaddr) public onlyOwner{
    require(BambooToken(_bambooaddr) == bamboo, "claimToken: invalid address");
    bamboo.claimOwnership();
}
```



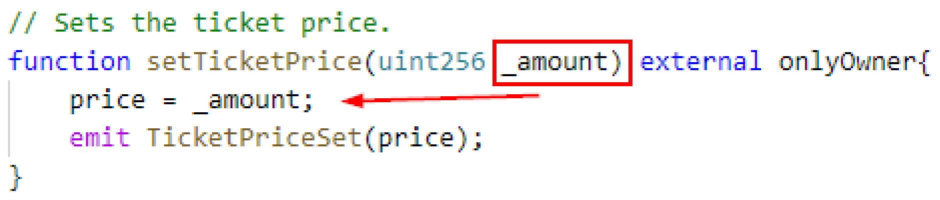
```
// Withdraw the bonus staking Bamboo available from this deposit.
function withdrawDailyBamboo(uint256 _depositId) public {
    BambooUserInfo storage user = bambooUserInfo[msg.sender];
    require(user.deposits[_depositId].active, "withdrawDailyBamboo: invalid id");
    uint256 depositEnd = _depositId.add(user.deposits[_depositId].lockTime);
    uint256 amount;
    uint256 ndays;
    (amount, ndays) = getClaimableBamboo(_depositId, msg.sender);
    uint256 newLastTime =  user.deposits[_depositId].lastTime.add(ndays.mul(86400));
    assert(newLastTime <= depositEnd);
    user.deposits[_depositId].lastTime =  newLastTime;
    // Mint the bonus bamboo
    bamboo.mint(address(this), amount);
    safeBambooTransfer(msg.sender, amount);
    emit BAMBOOBonusWithdraw(msg.sender, _depositId, amount, ndays);
}
```

- In order to avoid that the owners have an advantageous position, it's recommended to eliminate the _*withUpdate* argument of the **ZooKeeper** contract. You can obtain more information about the problem that this argument entails in the DraculaProtocol review.
- The **BambooDefi** project derives the functionality of its pools from the UniSwap project, which in its version 2 has included specific methods to support token swaps with fees in the transfers. We must highlight that BambooDeFi has removed this functionality from the **UniswapV2Router02** contract.

BD005. Lack of Inputs Validation

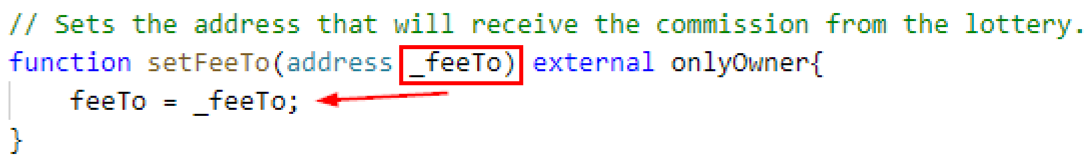
Some methods of the different contracts in the Bamboo DeFI project do not properly check the arguments, which can lead to major errors. Below we list the most significant examples.

- The *setTicketPrice* method of the **Raindrop** and **BBYP** contracts allow its value to be 0, which would alter the expected logic of the lottery. It is recommended to verify that the price is among the expected values of the business logic.



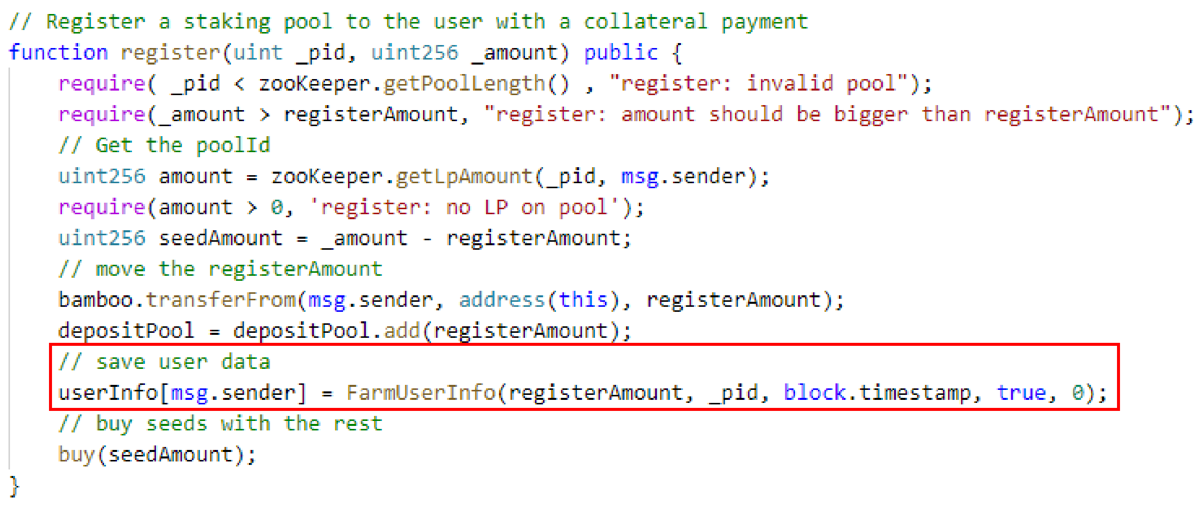
```
// Sets the ticket price.
function setTicketPrice(uint256 _amount) external onlyOwner{
    price = _amount;
    emit TicketPriceSet(price);
}
```

- In **Raindrop** smart contract the *setFeeTo* method allows the address to be established to *0x0000…* which would burn fees, it is recommended to previously verify the address.



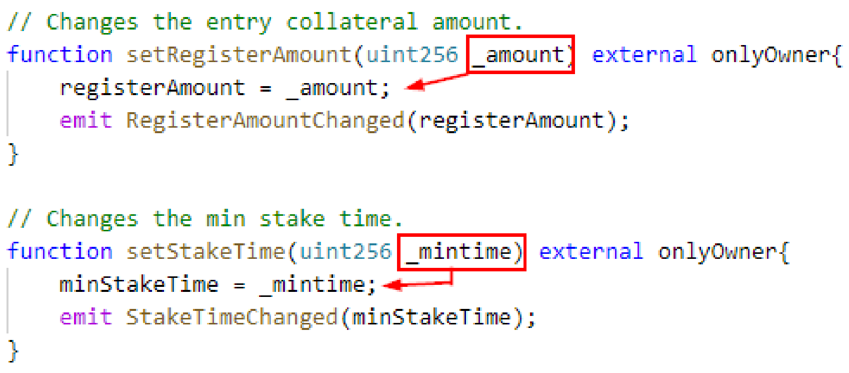
```
// Sets the address that will receive the commission from the lottery.
function setFeeTo(address _feeTo) external onlyOwner{
    feeTo = _feeTo;
}
```

- The *register* method of the **BambooField** smart contract does not verify if the user is already registered, overwriting the values. It should be previously checked if the user is registered before overwriting the user's information.



```
// Register a staking pool to the user with a collateral payment
function register(uint _pid, uint256 _amount) public {
    require( _pid < zooKeeper.getPoolLength() , "register: invalid pool");
    require(_amount > registerAmount, "register: amount should be bigger than registerAmount");
    // Get the poolId
    uint256 amount = zooKeeper.getLpAmount(_pid, msg.sender);
    require(amount > 0, 'register: no LP on pool');
    uint256 seedAmount = _amount - registerAmount;
    // move the registerAmount
    bamboo.transferFrom(msg.sender, address(this), registerAmount);
    depositPool = depositPool.add(registerAmount);
    // save user data
    userInfo[msg.sender] = FarmUserInfo(registerAmount, _pid, block.timestamp, true, 0);
    // buy seeds with the rest
    buy(seedAmount);
}
```

- In **BambooField** smart contract the methods *setRegisterAmount* and *setStakeTime* do not make any sort of verification. It is advisable to check that the received values are as expected.



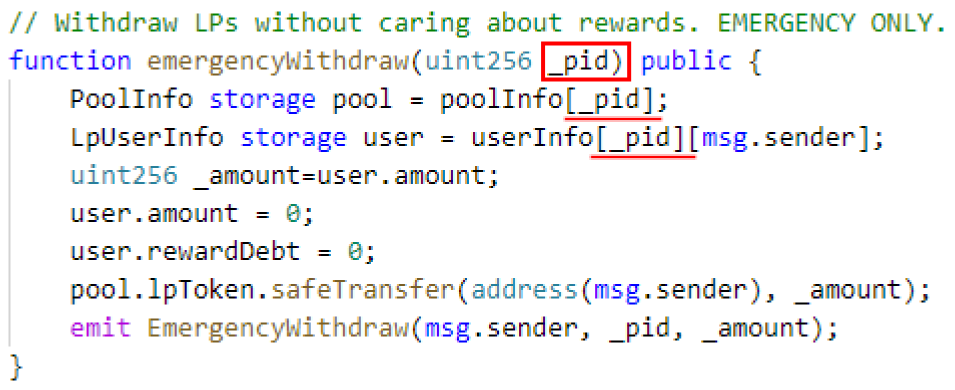
```
// Changes the entry collateral amount.
function setRegisterAmount(uint256 _amount) external onlyOwner{
    registerAmount = _amount;
    emit RegisterAmountChanged(registerAmount);
}

// Changes the min stake time.
function setStakeTime(uint256 _mintime) external onlyOwner{
    minStakeTime = _mintime;
    emit StakeTimeChanged(minStakeTime);
}
```

- The **ZooKeeper's** constructor does not accurately check the arguments, where a value inferior to 1 would break the contract's logic in *bambooPerBlock***.**

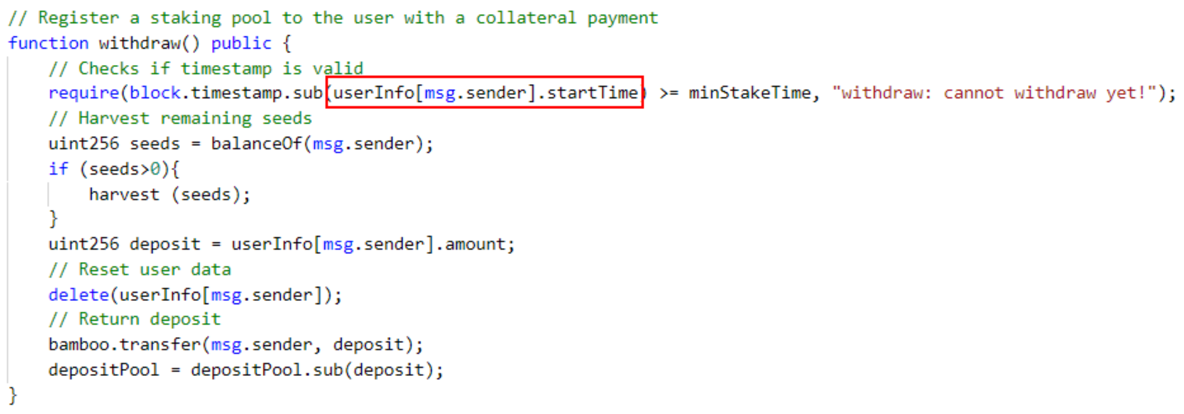
- The *addYieldMultiplier* function does not check that the values are not 0, which causes the *getYieldMultiplier* to return a *0* blocking the normal logic of all the functions that use it.

- The function *emergencyWithdraw* of **ZooKeeper** smart contract does not check that the *_pid* exists.



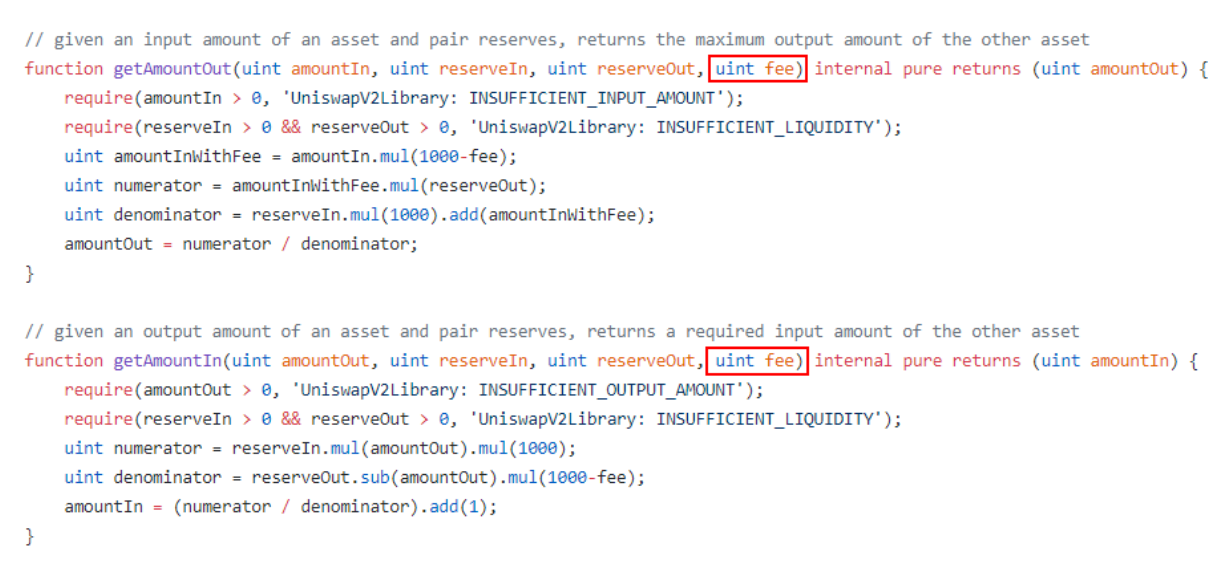
```
// Withdraw LPs without caring about rewards. EMERGENCY ONLY.
function emergencyWithdraw(uint256 _pid) public {
    PoolInfo storage pool = poolInfo[_pid];
    LpUserInfo storage user = userInfo[_pid][msg.sender];
    uint256 _amount=user.amount;
    user.amount = 0;
    user.rewardDebt = 0;
    pool.lpToken.safeTransfer(address(msg.sender), _amount);
    emit EmergencyWithdraw(msg.sender, _pid, _amount);
}
```

- In the **BambooField** contract, the *userInfo* mapping is used to store the user's registrations, but when this mapping is queried, it is not verified that the registration exists. When a registration does not exist, the default values will be returned and unexpected results may be obtained. This behavior can be observed in the *harvest* and *withdraw* functions.



```
// Register a staking pool to the user with a collateral payment
function withdraw() public {
    // Checks if timestamp is valid
    require(block.timestamp.sub(userInfo[msg.sender].startTime) >= minStakeTime, "withdraw: cannot withdraw yet!");
    // Harvest remaining seeds
    uint256 seeds = balanceOf(msg.sender);
    if (seeds>0){
        harvest (seeds);
    }
    uint256 deposit = userInfo[msg.sender].amount;
    // Reset user data
    delete(userInfo[msg.sender]);
    // Return deposit
    bamboo.transfer(msg.sender, deposit);
    depositPool = depositPool.sub(deposit);
}
```

- The **UniswapV2Library.sol** contract has been modified with respect to the original **Uniswap** contract in order to establish a *fee*, however throughout the modifications made in that fork it is established that the *fee* must be less than or equal to *50*.



```
// given an input amount of an asset and pair reserves, returns the maximum output amount of the other asset
function getAmountOut(uint amountIn, uint reserveIn, uint reserveOut, uint fee) internal pure returns (uint amountOut) {
    require(amountIn > 0, 'UniswapV2Library: INSUFFICIENT_INPUT_AMOUNT');
    require(reserveIn > 0 && reserveOut > 0, 'UniswapV2Library: INSUFFICIENT_LIQUIDITY');
    uint amountInWithFee = amountIn.mul(1000-fee);
    uint numerator = amountInWithFee.mul(reserveOut);
    uint denominator = reserveIn.mul(1000).add(amountInWithFee);
    amountOut = numerator / denominator;
}

// given an output amount of an asset and pair reserves, returns a required input amount of the other asset
function getAmountIn(uint amountOut, uint reserveIn, uint reserveOut, uint fee) internal pure returns (uint amountIn) {
    require(amountOut > 0, 'UniswapV2Library: INSUFFICIENT_OUTPUT_AMOUNT');
    require(reserveIn > 0 && reserveOut > 0, 'UniswapV2Library: INSUFFICIENT_LIQUIDITY');
    uint numerator = reserveIn.mul(amountOut).mul(1000);
    uint denominator = reserveOut.sub(amountOut).mul(1000-fee);
    amountIn = (numerator / denominator).add(1);
}
```

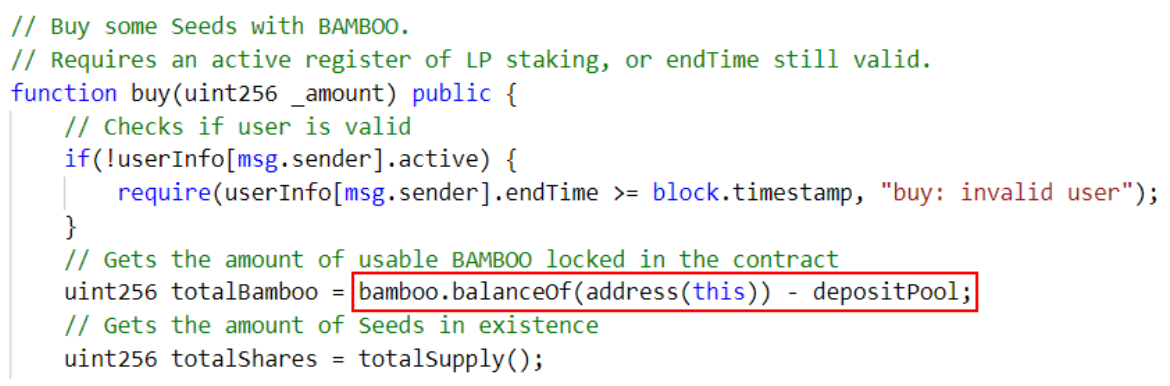
It is convenient to add the same verifications in the *getAmountOut* and *getAmountIn* functions to prevent fees greater than or equal to 1000 from triggering unexpected results.

```
require(fee <= 50, 'UniswapV2: INVALID_FEE');
```

BD006. Use of SafeMath

The OpenZeppelin SafeMath class is properly used throughout the contract to protect the contract from incorrect or malicious arithmetic operations. However, the **BambooField** contract performs a direct subtraction with the arithmetic operator instead of using safeMath's *sub* safe method.

An example of this issue can be found in: **BambooField.sol:64**
It should be noted that, although it is a good practice, the current implementation is safe and has a lower consumption of GAS, if the staked token works as expected.



```
// Buy some Seeds with BAMBOO.
// Requires an active register of LP staking, or endTime still valid.
function buy(uint256 _amount) public {
    // Checks if user is valid
    if(!userInfo[msg.sender].active) {
        require(userInfo[msg.sender].endTime >= block.timestamp, "buy: invalid user");
    }
    // Gets the amount of usable BAMBOO locked in the contract
    uint256 totalBamboo = bamboo.balanceOf(address(this)) - depositPool;
    // Gets the amount of Seeds in existence
    uint256 totalShares = totalSupply();
```

Source references

- BambooField.sol: 46, 64

BD007. Outdated Compiler Version

Solc frequently launches new versions of the compiler. Using an outdated version of the compiler can be problematic, especially if there are errors that have been made public or known vulnerabilities that affect this version.

We have detected that the audited contract uses the following version of Solidity pragma *0.6.12*:

It is always of good policy to use the most up to date version of the pragma.

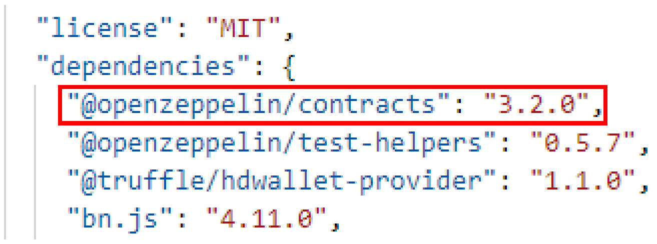
Reference

- https://github.com/ethereum/solidity/blob/develop/Changelog.md

BD008. Outdated Third-Party Libraries

The smart contracts analyzed inherit functionalities from open-zeppelin contracts that have been labeled obsolete and/or outdated; this does not imply a vulnerability by itself, because their logic does not present them, but it does imply that an update is not carried out by third party packages or libraries.

Currently the latest version of OpenZeppelin contracts is *3.3.0* therefore it would be convenient to include it as a reference instead of including the sources, in this way we will keep the development environment updated. In fact, the project includes a reference of the OpenZeppelin contracts in the *3.2.0* version, even though it is outdated and is not always used.



```
"license": "MIT",
"dependencies": {
  "@openzeppelin/contracts": "3.2.0",
  "@openzeppelin/test-helpers": "0.5.7",
  "@truffle/hdwallet-provider": "1.1.0",
  "bn.js": "4.11.0",
```

Additionally, these OpenZeppelin contracts are under the MIT license, which requires its license/copyright to be included within the code.

By using the original sources, in case the project resolves any vulnerability or bug in the code, you would obtain this update automatically. Consequently, it will avoid inheriting known vulnerabilities.

References

- https://github.com/OpenZeppelin/openzeppelin-contracts/releases

Recommendations

- Include third-party codes by package manager in all possible cases.
- Include in BambooDeFi project any references/copyright to OpenZeppelin code since it is under MIT license.

BD009. GAS Optimization

Software optimization is the process of modifying a software system to make an aspect of it work more efficiently or use less resources. This premise must be applied to smart contracts as well, so that they execute faster or in order to save GAS.

On Ethereum blockchain, GAS is an execution fee which is used to compensate miners for the computational resources required to power smart contracts. If the network usage is increasing, so will the value of GAS optimization.

These are some of the requirements that must be met to reduce GAS consumption:

- Short-circuiting.
- Remove redundant or dead code.
- Delete unnecessary libraries.
- Explicit function visibility.
- Use of proper data types.
- Use hard-coded CONSTANT instead of state variables.
- Avoid expensive operations in a loop.
- Pay special attention to mathematical operations and comparisons.

Redundant Code

The **getPoolLength** and **poolLength** methods expose the same value in the **ZooKeeper** contract, this produces an extra gas expense during the contract's deployment. Also, the code publicly exposes two methods in the ABI which return the same results.

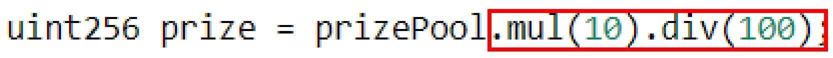
Source reference

- ZooKeeper.sol: 495

Constant operation cost

The execution of mathematical operations that return a constant value are usually optimized by the compiler, however, in solidity these operations are usually a result of calls made to the **SafeMath** methods, therefore they won't be optimized; and if they are found throughout the code they may produce an unnecessary expense to the user.

The *drawWinners* method in **Raindrop** executes a multiplication (by 10) and consequently a division (by 100), the same result can be obtained by directly dividing by 10 and this operation will also result in GAS saving.



```
uint256 prize = prizePool.mul(10).div(100);
```

Source references

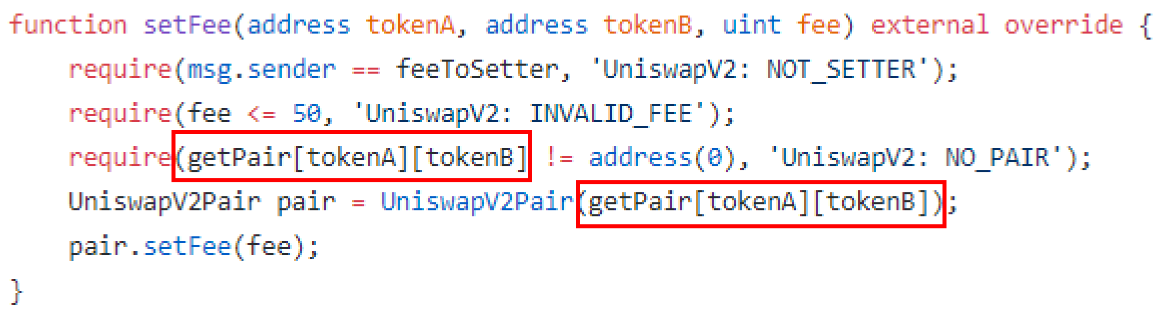
- Raindrop.sol: 131

Variable Caching

The access to storage in solidity is an extremely expensive process, so it should always be optimized as much as possible. Through caching variables, we can avoid querying the storage[2]; or through '*storage*' keyword we can obtain a pointer, both actions will help to have lower cost transactions.

In this way, in the **Raindrop** smart contract, by caching the result of access to the *participants[i]* storage on the *refund* method, a considerable gas saving can be obtained.

The same applies to the *setFee* function of the **UniswapV2Factory** contract, where a call to the *getPair* method can be eliminated by caching its value, as we have seen in the previous case. Since the method makes several calls to storage, it is possible to obtain a lower execution cost.



```
function setFee(address tokenA, address tokenB, uint fee) external override {
    require(msg.sender == feeToSetter, 'UniswapV2: NOT_SETTER');
    require(fee <= 50, 'UniswapV2: INVALID_FEE');
    require(getPair[tokenA][tokenB] != address(0), 'UniswapV2: NO_PAIR');
    UniswapV2Pair pair = UniswapV2Pair(getPair[tokenA][tokenB]);
    pair.setFee(fee);
}
```

Source references

- Raindrop.sol: 208-210
- UniswapV2Factory.sol: 67-68

Save contract calls

The calls made between solidity contracts have an extra cost, which is superior to a call made to a method in our own contract, this results in a significant benefit in terms of execution costs when optimizing each external cost[3].

[2] https://docs.soliditylang.org/en/v0.5.12/types.html
[3] https://www.ethervm.io/#F1

The *buyTickets* method of the **Raindrop** and **BBYP** contracts could save GAS if only one transaction is made (*safeTransfer*), instead of making one transaction by each iteration of the for loop.

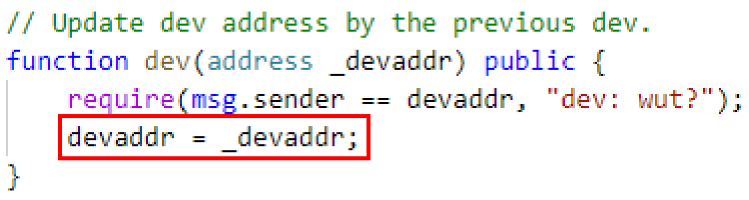
The **ZooKeeper** contract implements the *withdrawDailyBamboo* method, in this method a *mint* is executed and afterwards a transfer of the sender's amount is made. The function could be optimized by performing the *mint* directly to the sender.

Source references

- Raindrop.sol: 85
- BBYP.sol: 84

Unused code

The *devaddr* variable, declared in the dev method which exists in the **ZooKeeper** contract is not used throughout the smart contract. It is advisable to remove the methods and variables that don't have any utility when saving GAS in the deploy, besides it would make the contract more user-friendly, by avoiding exposing methods that are never used throughout the contract.



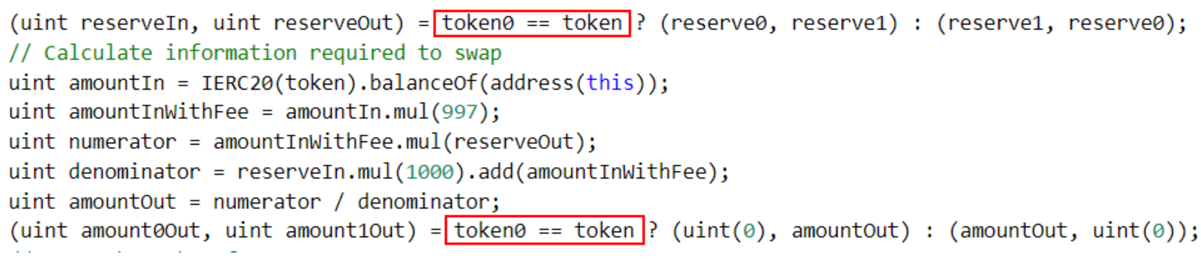
```
// Update dev address by the previous dev.
function dev(address _devaddr) public {
    require(msg.sender == devaddr, "dev: wut?");
    devaddr = _devaddr;
}
```

Source references

- ZooKeeper.sol: 93, 504

Logic Optimizations

Several conditionals of the **BambooFarmer** contract can be optimized with their corresponding GAS saving, these conditionals are in the *_toWETH* (#63, #70) and *_toBAMBOO* (#83, #89) methods. The condition for the initialization of such variables is the same for both cases, therefore it would be convenient to reuse the conditional when initializing all the necessary variables.



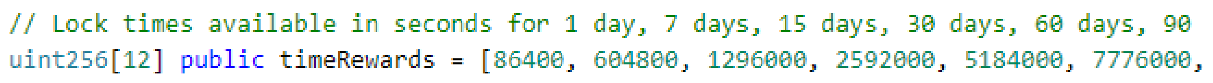
```
(uint reserveIn, uint reserveOut) = token0 == token ? (reserve0, reserve1) : (reserve1, reserve0);
// Calculate information required to swap
uint amountIn = IERC20(token).balanceOf(address(this));
uint amountInWithFee = amountIn.mul(997);
uint numerator = amountInWithFee.mul(reserveOut);
uint denominator = reserveIn.mul(1000).add(amountInWithFee);
uint amountOut = numerator / denominator;
(uint amount0Out, uint amount1Out) = token0 == token ? (uint(0), amountOut) : (amountOut, uint(0));
```

Source reference

- BambooFarmer.sol: 63,70, 83,89

BD010. Solidity Literals

In order to make the code easier to read and to minimize human errors, Solidity recommends the use of literals which consequently makes it more user friendly. It is possible to improve the reading of dates by using time units: Suffixes like *seconds*, *minutes*, *hours*, *days* and *weeks*, which are less prone to errors.



```
// Lock times available in seconds for 1 day, 7 days, 15 days, 30 days, 60 days, 90
uint256[12] public timeRewards = [86400, 604800, 1296000, 2592000, 5184000, 7776000,
```

References

- https://docs.soliditylang.org/en/latest/units-and-global-variables.html#time-units

Code References

- Raindrop.sol: 182,230
- BBYP.sol: 121,181
- ZooKeeper.sol: 36,427,429,443,445,448

BD011. Code Style

It has been possible to verify that, despite good quality code, there is a lack of order and structure that makes reading and analyzing the code difficult.

This is a very common bad practice, especially in these types of projects that are

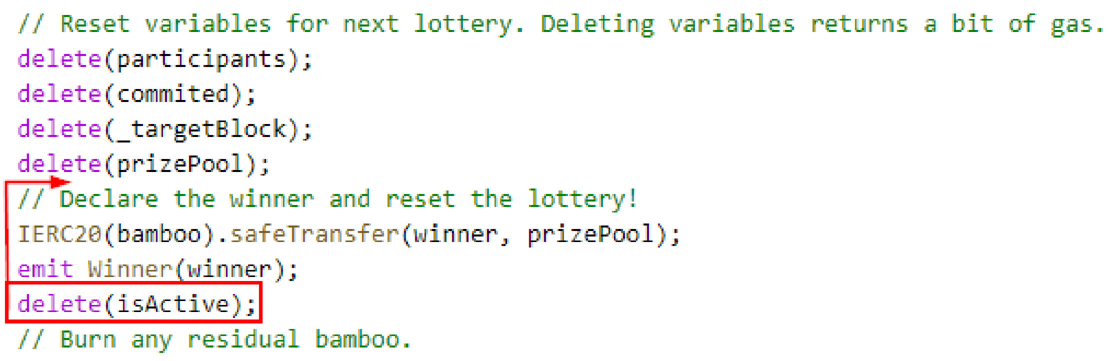
continually changing and improving. This is not a vulnerability, but it helps to improve the code and reduce the appearance of new vulnerabilities.
As a reference, it is always advisable to apply some coding style/good practices that can be found in multiple standards such as:

- "Solidity Style Guide" (https://docs.soliditylang.org/en/v0.8.0/style-guide.html).

These references are very useful to improve smart contract quality. Some of those practices are common and a popular accepted way to develop software.

Following, we detail a few points that could be improved in terms of style, quality, and readability of the code throughout the audited contracts.

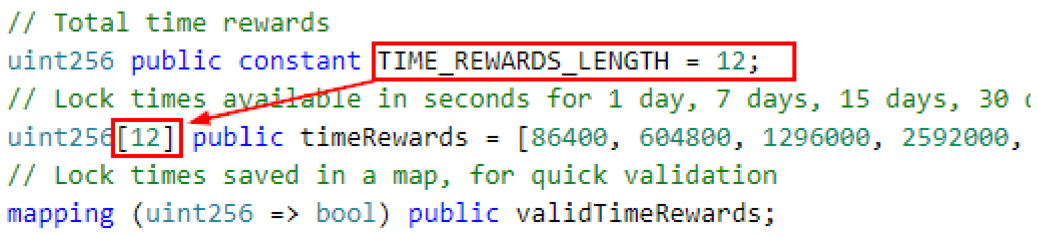
- The **Raindrop** and **BBYP** contracts implement a very similar logic and share many methods. It would be advisable to design a parent class with all the common logic, a more appropriate design would use inheritance, through which the quality of these contracts would be greatly improved.
- The project has dependencies on other projects, such as *wETH*. These have been added to the project, even though the best option is to make references. In case mocks of these contracts are needed, it is better for unit tests to place them in the test's folders separated from the project's code.
- In the **BBYP** smart contract, the *revealWinner* function has a block where it restarts a set of variables, however *isActive* is in a different block. It is recommended to group them so that the code is more readable.



```
// Reset variables for next lottery. Deleting variables returns a bit of gas.
delete(participants);
delete(commited);
delete(_targetBlock);
delete(prizePool);
// Declare the winner and reset the lottery!
IERC20(bamboo).safeTransfer(winner, prizePool);
emit Winner(winner);
delete(isActive);
// Burn any residual bamboo.
```

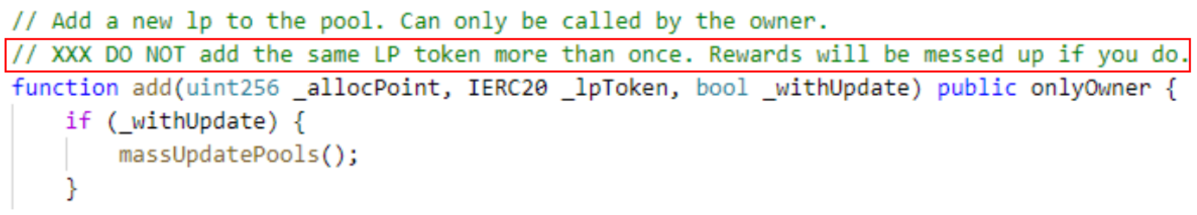
- In the **ZooKeeper** smart contract an array is declared that stores the times for the rewards, however, it does not use the constant

TIME_REWARDS_LENGTH that defines the size of said array. This could prevent a human error in an update where the array is out of sync regarding the constant.



```
// Total time rewards
uint256 public constant TIME_REWARDS_LENGTH = 12;
// Lock times available in seconds for 1 day, 7 days, 15 days, 30 (
uint256[12] public timeRewards = [86400, 604800, 1296000, 2592000,
// Lock times saved in a map, for quick validation
mapping (uint256 => bool) public validTimeRewards;
```

- In the *add* method of the **ZooKeeper** smart contract, there is a sensitive comment that could decrease the user's and the investor's confidence in the project, since the indicated problem in the comment seems to be already solved.



```
// Add a new lp to the pool. Can only be called by the owner.
// XXX DO NOT add the same LP token more than once. Rewards will be messed up if you do.
function add(uint256 _allocPoint, IERC20 _lpToken, bool _withUpdate) public onlyOwner {
    if (_withUpdate) {
        massUpdatePools();
    }
```

RED4SEC

Invest in Security, invest in your future