WildLands BitMaster

smart contracts
final audit report

March 2023

hashex.org

contact@hashex.org

Contents

1. Disclaimer

This is a limited report on our findings based on our analysis, in accordance with good industry practice at the date of this report, in relation to cybersecurity vulnerabilities and issues in the framework and algorithms based on smart contracts, the details of which are set out in this report. In order to get a full view of our analysis, it is crucial for you to read the full report. While we have done our best in conducting our analysis and producing this report, it is important to note that you should not rely on this report and cannot claim against us on the basis of what it says or doesn't say, or how we produced it, and it is important for you to conduct your own independent investigations before making any decisions. We go into more detail on this in the disclaimer below – please make sure to read it in full.

By reading this report or any part of it, you agree to the terms of this disclaimer. If you do not agree to the terms, then please immediately cease reading this report, and delete and destroy any and all copies of this report downloaded and/or printed by you. This report is provided for information purposes only and on a non-reliance basis and does not constitute investment advice. No one shall have any right to rely on the report or its contents, and HashEx and its affiliates (including holding companies, shareholders, subsidiaries, employees, directors, officers, and other representatives) (HashEx) owe no duty of care towards you or any other person, nor does HashEx make any warranty or representation to any person on the accuracy or completeness of the report. The report is provided "as is", without any conditions, warranties, or other terms of any kind except as set out in this disclaimer, and HashEx hereby excludes all representations, warranties, conditions, and other terms (including, without limitation, the warranties implied by law of satisfactory quality, fitness for purpose and the use of reasonable care and skill) which, but for this clause, might have effect in relation to the report. Except and only to the extent that it is prohibited by law, HashEx hereby excludes all liability and responsibility, and neither you nor any other person shall have any claim against HashEx, for any amount or kind of loss or damage that may result to you or any other person (including without limitation, any direct, indirect, special, punitive, consequential or pure economic loss or damages, or any loss of income, profits, goodwill, data, contracts, use of money, or business interruption, and whether in delict, tort (including without limitation negligence), contract, breach of statutory duty, misrepresentation (whether innocent or negligent) or otherwise under any claim of any nature whatsoever in any jurisdiction) in any way arising from or connected with this report and the use, inability to use or the results of the use of this report, and any reliance on this report. The analysis of the security is purely based on the smart contracts alone. No applications or operations were reviewed for security. No product code has been reviewed. HashEx owns all copyright rights to the text, images, photographs, and other content provided in the following document. When using or sharing partly or in full, third parties must provide a direct link to the original document mentioning the author (hashex.org).

2. Overview

HashEx was commissioned by the WildLands team to perform an audit of their smart contract. The audit was conducted between 13/02/2023 and 16/02/2023.

The purpose of this audit was to achieve the following:

- Identify potential security issues with smart contracts
- Formally check the logic behind given smart contracts.

Information in this report should be used for understanding the risk exposure of smart contracts, and as a guide to improving the security posture of smart contracts by remediating the issues that were identified.

The code is available at @wildlandsme/smart_contracts GitHub repository after the 9f5e7e3 commit. Only BitGold, BitRAM, and BITMaster contracts were in scope of this audit.

Whitepaper is available in the WildLands team Medium blog.

Update. The WildLands team has responded to this report, the updated code is located in the same repository after the d57d7bf commit.

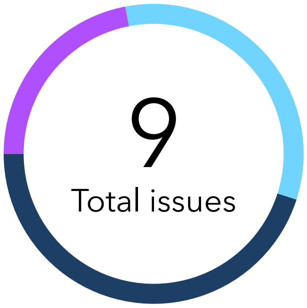
2.1 Summary

Project name	WildLands BitMaster
URL	https://wildlands.me
Platform	Ethereum
Language	Solidity

2.2 Contracts

Name	Address
BitGold	`0x2b1E1E8e1D821564016bBC4383D1159cd37b06A4`
BitRAM	`0x04CB8274aC5135491332B8115217B36D6c7D40b5`
BITMaster	`0x386E2d431429589168f464758a8Fb2d18b1E7b8D`

3. Found issues



Medium	2 (22%)
Low	3 (33%)
Info	4 (45%)

C3. BITMaster

ID	Severity	Title	Status
C3-01	Medium	Emergency withdrawal is not possible	Resolved
C3-02	Medium	Block numbers are used as time equivalent	Resolved
C3-03	Low	Unfair distribution of awards without massUpsatePool()	Acknowledged
C3-04	Low	Possible reward dilution	Resolved
C3-05	Low	Gas optimizations	Partially fixed
C3-06	Info	Whitelisting and fee excluding are irreversible	Resolved
C3-07	Info	Typos	Partially fixed
C3-08	Info	Lack of documentation for burning fees	Resolved
C3-09	Info	Checking of duplicating pools	Resolved

4. Contracts

C1. BitGold

Overview

An ERC-20 standard token built on top of OpenZeppelin's implementation. Allows minting from a single owner address to be set as a BITMaster contract without the possibility of further transfers of ownership.

No issues were found.

C2. BitRAM

Overview

A simple Ownable contract to hold selected ERC-20 token (BitGold) and transfer it upon owner's request. The ownership is meant to be given to the BITMaster contract.

No issues were found.

C3. BITMaster

Overview

A Masterchef-like (see SushiSwap's version) contract that allows users to provide liquidity in form of selected pools of ERC-20 tokens in order to farm minting rewards. Several features are introduced into the BITMaster contract: pools may have deposit and withdraw fees up to 10% (going to treasury), part of this fees may be burned, withdrawing is not possible until the locking period is cleared (individual and optional for each pool, can't exceed 30 days), referral system uses external WildlandsMemberCards NFT contract to acquire part of deposit/withdraw fees.

Issues

C3-01 Emergency withdrawal is not possible ● Medium ✓ Resolved

Emergency withdrawal doesn't work in case of an emergency since it requires clearance of the locking period. Its only use is to withdraw staked funds without collecting the rewards.

```
function emergencyWithdraw(uint256 _pid) ... {
  require(timeToUnlock(_pid, msg.sender) == 0, "withdraw: tokens are still locked.");
  ...
}
```

Recommendation

Consider removing the `emergencyWithdraw()` function to eliminate possible risks of abuse since it doesn't secure users' funds.

C3-02 Block numbers are used as time equivalent ● Medium ✓ Resolved

Whitepaper tokenomics is described in terms of halving the rewards every year. The BITMaster contract uses a constant 12 second-per-block block production time, meaning the actual rewards distribution depends on the Ethereum network's future state.

```
uint256 public UNITS_PER_DAY = 86400 / 12; // 7200 blocks per day

uint256 public bitPerBlock = (10 * 10 ** 6 - 145000) * DECIMALS_TOKEN / (2 * 365 *
UNITS_PER_DAY);

function applyHalfing(uint256 _amount, uint256 testcounter) ... {
  ...
  uint256 blocks = block.number.add(testcounter).sub(startBlock);
  uint256 i = blocks.div(UNITS_PER_DAY.mul(365));
  return _amount.div(2**i);
}
```

Recommendation

Consider moving to timestamps or modifying the documentation.

C3-03 Unfair distribution of awards without massUpsatePool()

Low | Acknowledged

The reward distribution for pools where the `_updatePool()` function is rarely called and can be re-distributed retrospectively if new pools are added or the allocation scheme updated without the `_withUpdate` flag. Moreover, the `set()` function allows updating the pool's allocation without updating the pool firsthand, meaning the reward distribution since the last update (`lastRewardBlock`) would be calculated using a new allocation.

C3-04 Possible reward dilution

Low | Resolved

The contract uses its contract token balance for the LP supply variable which should be equal to the sum of users' deposits. If someone accidentally sends tokens to the contract and the contract's balance in tokens is bigger than the sum of deposits and the rewards will be diluted.

```
    function _updatePool(uint256 _pid) internal validatePool(_pid) {
        PoolInfo storage pool = poolInfo[_pid];
        if (block.number <= pool.lastRewardBlock) {
            return;
        }
        uint256 lpSupply = pool.stakeToken.balanceOf(address(this));
        ...
            pool.accBitsPerShare =
pool.accBitsPerShare.add(bitReward.mul(DECIMALS_SHARE_REWARD).div(lpSupply));
        ...
    }
```

Recommendation

Add an additional `lpSupply` variable to the PoolInfo structure which tracks the sum of users' deposited tokens to the pool.

C3-05 Gas optimizations

Low | Partially fixed

1. `PoolInfo` struct should be rearranged to save gas: the `stakeToken` address should be packed with `uint16` and `bool` variables to fit in a single storage slot.
2. The variables `MAX_PERCENT`, `DECIMALS_TOKEN`, `DECIMALS_SHARE_REWARD`, `DEAD_ADDRESS`, `UNITS_PER_DAY`, `bitPerBlock` should be declared as constants.
3. A single variable for total allocation points in all pools other than the first pool should be used in order to reduce gas cost for the `updateStakingPool()` function.
4. Possible double read from the storage of the `startBlock` variable in the `add()` function, `poolInfo[0].allocPoint` in the `updateStakingPool()` function, `pool.lastRewardBlock` and `totalAllocPoint` in the `pendingBit()` function, `pool.depositFeeBP` in the `deposit()`, `pool.withdrawFeeBP` in the `emergencyWithdraw()` function.
5. Double read from the storage of `pool.lastRewardBlock` variable in the `_updatePool()` function, `affiliatee[msg.sender]` and `wildlandcard.getTokenIdByCode()` in the `handleFee()` function.
6. Unnecessary read from the storage of `user.amount` and `pool.lockTimer` variables in the `deposit()` function.
7. Multiple reads from the storage of `user.amount`and `pool.withdrawFeeBP` variables in the `withdraw()` function, `poolInfo.length` in the `setStartblock()` function.
8. Unchecked math could be used in the `applyHalfing()`, `timeToUnlock()` functions.
9. Unnecessary `mul/div` operations with `DECIMALS_SHARE_REWARD` in the `pendingBit()` function.
10. Redundant call to `getMultiplier()`, which returns 1 with given parameters, in the `mintingInfo()` function.

C3-06 Whitelisting and fee excluding are irreversible

Info | Resolved

The `whiteListAddress()` and `excludeFromFees()` governance functions work one-way, making any mistakes permanent.

```
function whiteListAddress(address _address) external onlyOwner {
  // whitelist addresses as members, such as partner contracts
  // cannot be undone as this might mess up the partners' contracts
  isWhiteListed[_address] = true;
  emit WhiteListed(_address);
}

function excludeFromFees(address _address) external onlyOwner {
  // whitelist addresses as non-fee-payers, such as partner contracts
  // cannot be undone as this might mess up the partners' contracts
  IsExcludedFromFees[_address] = true;
  emit ExcludedFromFees(_address);
}
```

C3-07 Typos

Info | Partially fixed

Typos reduce the code's readability. Typos in 'secion', 'affiliatee', 'stakable', 'halfing', 'meber', 'mechansims', 'affilite', 'affiliator', 'infintity', 'depost', 'neded', 'minues', 'tresaruy', 'addresse'.

C3-08 Lack of documentation for burning fees

Info | Resolved

The `burnDepositFeeBP` and `burnWithdrawFeeBP` fields of `PoolInfo` structure are meant to be nominated in basis points, but the actual burning amount is calculated against `depositFeeBP`/ `withdrawFeeBP`.

```
struct PoolInfo {
  IERC20 stakeToken;
  ...
  uint16 depositFeeBP;
  uint16 burnDepositFeeBP;
  uint16 withdrawFeeBP;
```

```
  uint16 burnWithdrawFeeBP;
  bool requireMembership;
}

uint32 public MAX_PERCENT = 1e4;

function deposit(uint256 _pid, uint256 _amount) ... {
  ...
  amount_fee = _amount.mul(pool.depositFeeBP).div(MAX_PERCENT);
  uint256 burn_fee = amount_fee.mul(pool.burnDepositFeeBP).div(pool.depositFeeBP);
  ...
}

function withdraw(uint256 _pid, uint256 _amount) ... {
  ...
  amount_fee = _amount.mul(pool.withdrawFeeBP).div(MAX_PERCENT);
  uint256 burn_fee = amount_fee.mul(pool.burnWithdrawFeeBP).div(pool.withdrawFeeBP);
  ...
}
```

C3-09 Checking of duplicating pools

Info — Resolved

Pool duplication is checked with the use of `poolExistence` boolean mapping, which explicitly denies pools with the same tokens.

```
mapping(IERC20 => bool) public poolExistence;

modifier nonDuplicated(IERC20 _lpToken) {
  require(poolExistence[_lpToken] == false, "add: existing pool");
  _;
}
```

There's a more universal way of allowing duplicating pools with a possible different allocation (some of them can be disabled) by accurately accounting for the stored pool's balances inside the `PoolInfo` structure.

```
struct PoolInfo {
```

```
  uint256 allocPoint;
  uint256 poolStaked;
  ...
  IERC20 stakeToken;
  ...
}

function _updatePool(uint256 _pid) ... {
  PoolInfo storage pool = poolInfo[_pid];
  uint256 lpSupply = pool.poolStaked;
  ...
}

function pendingBit(uint256 _pid, address _user) ... {
  PoolInfo storage pool = poolInfo[_pid];
  uint256 lpSupply = pool.poolStaked;
  ...
}
```

5. Conclusion

2 medium, 3 low severity issues were found during the audit. 2 medium, 1 low issues were resolved in the update.

The audited contracts are highly dependent on the owner’s account. Users using the project have to trust the owner and that the owner's account is properly secured.

This audit includes recommendations on code improvement and the prevention of potential attacks.

Appendix A. Issues' severity classification

- **Critical.** Issues that may cause an unlimited loss of funds or entirely break the contract workflow. Malicious code (including malicious modification of libraries) is also treated as a critical severity issue. These issues must be fixed before deployments or fixed in already running projects as soon as possible.
- **High.** Issues that may lead to a limited loss of funds, break interaction with users, or other contracts under specific conditions. Also, issues in a smart contract, that allow a privileged account the ability to steal or block other users' funds.
- **Medium.** Issues that do not lead to a loss of funds directly, but break the contract logic. May lead to failures in contracts operation.
- **Low.** Issues that are of a non-optimal code character, for instance, gas optimization tips, unused variables, errors in messages.
- **Informational.** Issues that do not impact the contract operation. Usually, informational severity issues are related to code best practices, e.g. style guide.

Appendix B. List of examined issue types

- Business logic overview
- Functionality checks
- Following best practices
- Access control and authorization
- Reentrancy attacks
- Front-run attacks
- DoS with (unexpected) revert
- DoS with block gas limit
- Transaction-ordering dependence
- ERC/BEP and other standards violation
- Unchecked math
- Implicit visibility levels
- Excessive gas usage
- Timestamp dependence
- Forcibly sending ether to a contract
- Weak sources of randomness
- Shadowing state variables
- Usage of deprecated code

contact@hashex.org

@hashex_manager

blog.hashex.org

linkedin

github

twitter

HashEx
BLOCKCHAIN SECURITY