AudiTBlock

PytBit

v0.8.20;

Low-Risk

Low-risk code

Medium-Risk

Medium-risk code

High-Risk

High-risk code

Executive Summary

Contract Name PytBit

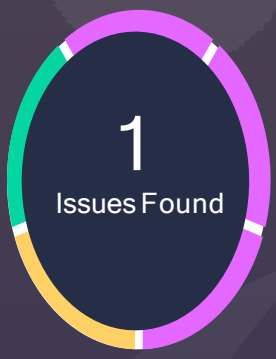
Overview

PytBit introduces a revolutionary concept within the decentralized finance (DeFi) arena, distinctly connecting Bitcoin with the broader cryptocurrency ecosystem. This project creatively generates tokens in alignment with Bitcoin block generation, leveraging the strength and market impact of Bitcoin to drive its unique token dynamics.

Method Manual Review, Functional Testing, Automated Testing etc.

Scope of Audit

The scope of this audit was to analyze the contract codebase for quality, security, and correctness.



High Medium Low Informational

	High	Medium	Low	Informational
Open Issues	0	0	0	0
Acknowledged Issues	0	1	0	0
Partially Resolved Issues	0	0	0	0
Resolved Issues	0	0	0	0

Smart Contract Weakness Classification (SWC) Vulnerabilities for Attacks

- Re-entrancy
- Timestamp Dependence
- Gas Limit and Loops
- Exception Disorder
- Gasless Send
- Use of tx.origin
- Compiler version not fixed
- Address hardcoded
- Divide before multiply
- Integer overflow/underflow
- Dangerous strict equalities
- Tautology or contradiction
- Missing Zero Address Validation
- Return values of low-level calls
- Revert/require functions
- Private modifier
- Using block.timestamp
- Multiple Sends
- Using SHA3
- Using suicide
- Using throw
- Using inline assembly

Types of Severities

High

A high-severity issue or vulnerability means that your smart contract can be exploited. Issues on this level are critical to the smart contract's performance or functionality, and we recommend these issues be fixed before moving to a live environment.

Medium

The issues marked as medium severity usually arise because of errors and deficiencies in the smart contract code. Issues on this level could potentially bring problems, and they should still be fixed.

Low

Low-level severity issues can cause minor impact and or are just warnings that can remain unfixed for now. It would be better to fix these issues at some point in the future.

Informational

These are severity issues that indicate an improvement request, a general question, a cosmetic or documentation error, or a request for information. There is low-to-no impact.

Techniques and Methods

The overall quality of code.

- Use of best practices.
- Code documentation and comments match logic and expected behavior.
- Token distribution and calculations are as per the intended behavior mentioned in the whitepaper.
- implementation of ERC-20 token standards.
- Efficient use of gas.
- Code is safe from re-entrance and other vulnerabilities.

The following techniques, methods, and tools were used to review all the smart contracts.

Structural Analysis

In this step, we have analyzed the design patterns and structure of smart contracts. A thorough check was done to ensure the smart contract is structured in a way that will not result in future problems.

Static Analysis

Static analysis of smart contracts was done to identify contract vulnerabilities. In this step, a series of automated tools are used to test the security of smart contracts.

Code Review / Manual Analysis

Manual analysis or review of code was done to identify new vulnerabilities or verify the vulnerabilities found during the static analysis. Contracts were completely manually analyzed, and their logic was checked and compared with the one described in the whitepaper. Besides, the results of the automated analysis were manually verified.

Gas Consumption

In this step, we have checked the behavior of smart contracts in production. Checks were done to know how much gas gets consumed and the possibilities of optimization of code to reduce gas consumption.

Tools and Platforms Used for Audit

Remix IDE, Truffle, Truffle Team, Solhint, Mythril, Slither, Solidity statistic analysis.

Phase 1

Project - PytBit

High Severity Issues:

No issues found

Medium Severity Issues:

No issues found

Low Severity Issues:

No issues found

Informational Severity Issues:

1. High Consuming Gas Fee

```
31      uint256 public constant MAX_SUPPLY = 21000000 * (10**18);
32      uint256 public lastMintTime;
33      uint256 public maxTokensPerBlock = 50 * (10**18);
34      uint256 public bitcoinPrice;
35      uint256 public taxRate = 5;
36      uint256 public blockCounter;
37      uint256 public blocksUntilHalving = 210000;
38      uint256 public halvingCounter;
39      mapping(address => Staker) public stakers;
40      mapping(address => uint256) public lastSellTime;
41      mapping(address => uint256) public amountSoldInWindow;
42      mapping(address => bool) public blacklist;
43      uint256 public sellLimit = 1000 * (10**18);
44      uint256 public sellWindow = 24 hours;
45      uint256 public massiveSellTax = 20;
46      uint256 public bitcoinVolume;
47      uint256 public adjustmentFactor = 100;
48      uint256 public basePrice = 50000 * (10**18);
49      uint256 public baseVolume = 1000 * (10**18);
50      uint256 public adjustedLiquidity;
51      uint256 public liquidityUnlockTime;
```

```
PROBLEMS  OUTPUT  TERMINAL  DEBUG CONSOLE  PORTS                    powershell

PytBit.calculateLoyaltyPoints(uint256,uint256) (contracts/PytBit.sol#496-507) performs a multiplication on the result of a division
:
        - stakingDuration = (endTime - startTime) / 86400 (contracts/PytBit.sol#498)
        - (stakingDuration * loyaltyMultiplier) / 100 (contracts/PytBit.sol#506)
PytBit.calculateLoyaltyPoints(uint256,uint256) (contracts/PytBit.sol#496-507) performs a multiplication on the result of a division
:
        - stakingDuration = (endTime - startTime) / 86400 (contracts/PytBit.sol#498)
        - loyaltyMultiplier += (stakingDuration * 10) / 30 (contracts/PytBit.sol#503)
Reference: https://github.com/crytic/slither/wiki/Detector-Documentation#divide-before-multiply
```

Description :

Our auditor found this issue manually. It is not a bug or vulnerability. We simply acknowledge that the contract used multiple storage type slots which consumed a high gas fee, and the contractor had too many declarations.

Recommendation:

The number of Storage slots and the ways you represent your data in your Smart Contract affect gas usage heavily. Here are some recommendations

- **Limiting Storage use:** Since Storage is the most expensive type of Memory, you need to use it as little as possible, which isn't always easy. Use Storage only for the essential data, nothing more. Try reducing the usage of blockchain Storage. For instance, transient (non-permanent) data can be stored in Memory.

- **Packing variables**: The minimum amount of Memory in Ethereum is a 256-bit slot. Even if slots aren't full, you need to pay for an integer quantity. To avoid this, you can use the variable packing technique.

- **Uint* vs. Uint256**: The EVM performs operations in 256-bit chunks, so converting a uint* (unsigned integers smaller than 256 bits) to uint256 consumes additional gas.
- **Mapping vs. array:** A list of data is only represented by two data types in Solidity: arrays and maps. Mappings are more efficient and less expensive, while arrays are inerrable and packable.
- Fixed size: Any fixed-size variable in Solidity is less expensive than a variable-size one. Use a fixed-size array instead of a dynamic one when possible.
- Default value: When variables are initialized, it’s a good practice to set their values, but this uses gas. In Solidity, all variables are set to 0 by default. Therefore, if a variable's default value is 0, don't explicitly set it to that value.
- Constants: For unchanging data, use constants over variables.
- Internal function calls: When you call a public function, it takes much longer than calling an internal one since all parameters are copied into the Memory.
- Fewer functions: Try to keep the number of internal and private functions to a minimum to balance function complexity and quantity. In turn, this will help you reduce gas fees upon execution by reducing the number of function calls.
- Short circuit: When it comes to logical expressions, try simplifying the complex ones as much as possible.
- Limiting modifiers: The code of modifiers is placed within a modified function, which increases its size and gas usage. To avoid this, reduce the number of modifiers.
- Single line swap: In one instruction, you can exchange the values of two variables.

Status

Informational

Phase 2

```
PytBit.updateAdjustedLiquidity() (contracts/PytBit.sol#220-225) performs
a multiplication on the result of a division:
        - volumeAdjustment = (bitcoinVolume * adjustmentFactor) /
baseVolume (contracts/PytBit.sol#223)
        - adjustedLiquidity = liquidity * priceAdjustment *
volumeAdjustment / (adjustmentFactor ** 2) (contracts/PytBit.sol#224)
PytBit.mint(uint256,uint256,uint256,uint256,bytes32)
(contracts/PytBit.sol#275-328) performs a multiplication on the result of
a division:
        - btcToTokenRate = btcPrice / 1000 (contracts/PytBit.sol#286)
        - btcMinerRewardInTokens = btcMinerReward * btcToTokenRate
(contracts/PytBit.sol#287)
PytBit.mint(uint256,uint256,uint256,uint256,bytes32)
(contracts/PytBit.sol#275-328) performs a multiplication on the result of
a division:
        - bonusTokens = (entropy + 1) * (timeElapsed / (24 * 3600)) * (1
- (totalSupply() / MAX_SUPPLY))
(contracts/PytBit.sol#290)PytBit.rewardHolders()
(contracts/PytBit.sol#448-486) performs a multiplication on the result of
a division:
        - totalRewards = (totalStaked * rewardRate) / 100
(contracts/PytBit.sol#470)
        - userReward = ((staker_scope_2.amountStaked * totalRewards) /
totalStaked) + baseBonus (contracts/PytBit.sol#478)
PytBit.rewardHolders() (contracts/PytBit.sol#448-486) performs a
multiplication on the result of a division:
        - userReward = (userReward * (loyaltyPoints_scope_3 +
totalLoyaltyPoints)) / totalLoyaltyPoints (contracts/PytBit.sol#479)
        - userReward = userReward * rewardMultiplier
(contracts/PytBit.sol#480)
PytBit.calculateLoyaltyPoints(uint256,uint256) (contracts/PytBit.sol#496-
507) performs a multiplication on the result of a division:
        - stakingDuration = (endTime - startTime) / 86400
(contracts/PytBit.sol#498)
        - (stakingDuration * loyaltyMultiplier) / 100
(contracts/PytBit.sol#506)
PytBit.calculateLoyaltyPoints(uint256,uint256) (contracts/PytBit.sol#496-
507) performs a multiplication on the result of a division:
        - stakingDuration = (endTime - startTime) / 86400
(contracts/PytBit.sol#498)
        - loyaltyMultiplier += (stakingDuration * 10) / 30
(contracts/PytBit.sol#503)
Reference: https://github.com/crytic/slither/wiki/Detector-
Documentation#divide-before-multiply
```

```
PytBit.stake(uint256) (contracts/PytBit.sol#419-430) uses a dangerous strict
equality:
        - staker.amountStaked == amount (contracts/PytBit.sol#426)
Reference: https://github.com/crytic/slither/wiki/Detector-
Documentation#dangerous-strict-equalities

PytBit.updateBitcoinData(uint256,uint256) (contracts/PytBit.sol#188-195) should
emit an event for:
        - bitcoinPrice = newPrice (contracts/PytBit.sol#192)
        - bitcoinVolume = newVolume (contracts/PytBit.sol#193)
PytBit.adjustTransactionFee(uint256) (contracts/PytBit.sol#200-216) should emit
an event for:
        - bitcoinPrice = newBitcoinPrice (contracts/PytBit.sol#215)
Reference: https://github.com/crytic/slither/wiki/Detector-
Documentation#missing-events-arithmetic

PytBit.releaseOwnerTokens() (contracts/PytBit.sol#121-126) uses timestamp for
comparisons
        Dangerous comparisons:
        - require(bool,string)(block.timestamp >= ownerLockTime,Tokens are
still locked) (contracts/PytBit.sol#122)
        - require(bool,string)(balanceOf(address(this)) >= OWNER_ALLOCATION,Not
enough tokens available for release) (contracts/PytBit.sol#123)
PytBit.unlockOwnerTokens() (contracts/PytBit.sol#139-150) uses timestamp for
comparisons
        Dangerous comparisons:
        - require(bool,string)(block.timestamp - ownerVestingStart >=
ownerVestingDuration,Vesting period has not ended) (contracts/PytBit.sol#141)
PytBit.withdrawLiquidity(uint256) (contracts/PytBit.sol#172-179) uses timestamp
for comparisons
        Dangerous comparisons:
        - require(bool,string)(block.timestamp >= liquidityUnlockTime,La
liquidita' e ancora bloccata) (contracts/PytBit.sol#173)
PytBit.mint(uint256,uint256,uint256,uint256,bytes32) (contracts/PytBit.sol#275-
328) uses timestamp for comparisons
        Dangerous comparisons:
        - require(bool,string)(totalSupply() + tokensToMint <=
MAX_SUPPLY,Superata la fornitura massima)
(contracts/PytBit.sol#293)PytBit.stake(uint256) (contracts/PytBit.sol#419-430)
uses timestamp for comparisons
        Dangerous comparisons:
        - require(bool,string)(balanceOf(msg.sender) >= amount,Saldo
insufficiente per lo staking) (contracts/PytBit.sol#421)
        - staker.amountStaked == amount (contracts/PytBit.sol#426)
PytBit.rewardHolders() (contracts/PytBit.sol#448-486) uses timestamp for
comparisons
        Dangerous comparisons:
        - require(bool,string)(totalLoyaltyPoints > 0,I punti fedelta totali
```

```
devono essere maggiori di zero) (contracts/PytBit.sol#465)
PytBit.calculateLoyaltyPoints(uint256,uint256) (contracts/PytBit.sol#496-507) uses
timestamp for comparisons
        Dangerous comparisons:
        - require(bool,string)(endTime >= startTime,Il tempo di fine deve essere
maggiore o uguale al tempo di inizio) (contracts/PytBit.sol#497)
        - stakingDuration > 30 (contracts/PytBit.sol#502)
Reference: https://github.com/crytic/slither/wiki/Detector-Documentation#block-
timestamp

Address._revert(bytes) (node_modules/@openzeppelin/contracts/utils/Address.sol#146-
158) uses assembly
        - INLINE ASM (node_modules/@openzeppelin/contracts/utils/Address.sol#151-
154)
Reference: https://github.com/crytic/slither/wiki/Detector-Documentation#assembly-
usage

PytBit.removeStaker(address) (contracts/PytBit.sol#409-417) has costly operations
inside a loop:
        - allStakers.pop() (contracts/PytBit.sol#413)
Reference: https://github.com/crytic/slither/wiki/Detector-Documentation#costly-
operations-inside-a-loop

(node_modules/@openzeppelin/contracts/utils/ReentrancyGuard.sol#4) necessitates a
version too recent to be trusted. Consider deploying with 0.8.18.
Pragma version^0.8.20 (contracts/PytBit.sol#2) necessitates a version too recent to
be trusted. Consider deploying with 0.8.18.
solc-0.8.20 is not recommended for deployment
Reference: https://github.com/crytic/slither/wiki/Detector-Documentation#incorrect-
versions-of-solidity

Low level call in Address.functionDelegateCall(address,bytes)
(node_modules/@openzeppelin/contracts/utils/Address.sol#104-107):
        - (success,returndata) = target.delegatecall(data)
(node_modules/@openzeppelin/contracts/utils/Address.sol#105)
Reference: https://github.com/crytic/slither/wiki/Detector-Documentation#low-level-
calls

Parameter PytBit.addStaker(address)._staker (contracts/PytBit.sol#405) is not in
mixedCase
Parameter PytBit.removeStaker(address)._staker (contracts/PytBit.sol#409) is not in
mixedCase
Reference: https://github.com/crytic/slither/wiki/Detector-
Documentation#conformance-to-solidity-naming-conventions

PytBit.lastMintTime (contracts/PytBit.sol#32) should be immutable
PytBit.ownerLockTime (contracts/PytBit.sol#59) should be immutable
Reference: https://github.com/crytic/slither/wiki/Detector-Documentation#state-
variables-that-could-be-declared-immutable

PytBit.taxRate (contracts/PytBit.sol#35) should be constant
PytBit.threshold (contracts/PytBit.sol#53) should be constant
PytBit.volatilityThreshold (contracts/PytBit.sol#198) should be constant
Reference: https://github.com/crytic/slither/wiki/Detector-Documentation#state-
variables-that-could-be-declared-constant
```

Phase 3

Functional Testing

Some of the tests performed are mentioned below:

- ✔ Should UnlockOwnerTokens With Owner vesting not initialized!.
- ✔ Should be mint if btc price > 0
- ✔ Should revert when none-owner calls onlyOwner methods
- ✔ Should work mint method correctly
- ✔ Should mint to correct supply
- ✔ Should releaseOwnerTokens with Tokens are still locked
- ✔ Should work transfer method correctly
- ✔ Should work transfer from method correctly
- ✔ Should update AdjustedLiquidity correctly
- ✔ Should Work all modifier
- ✔ Should add a user to addToBlacklist
- ✔ Should remove user from removeFromBlacklist
- ✔ Should be Struct type bitcoinBlockDetails

Closing Summary

Security Assessment Summary:

A rigorous security assessment of pytbit was undertaken, meticulously following the established audit procedures. This comprehensive evaluation uncovered several vulnerabilities of varying severity levels, which have been carefully categorized. To strengthen code robustness and elevate the overall security posture, a set of actionable recommendations and best practices have been meticulously formulated. The development team has wholeheartedly acknowledged all identified issues and is steadfastly committed to addressing them expeditiously.

This enhanced version of the text provides additional value by:

- Emphasizing the rigor and thoroughness of the security assessment.
- Highlighting the meticulousness with which vulnerabilities were identified and categorized.
- Emphasizing the comprehensiveness of the recommendations and best practices provided.
- Underscoring the development team's commitment to addressing the identified issues promptly.

By incorporating these enhancements, the text conveys a stronger sense of the security assessment's thoroughness, the significance of the identified vulnerabilities, and the team's dedication to ensuring Pytbit's security.

AuditBlock

AuditBlock is a blockchain security company that provides professional services and solutions for securing blockchain projects. They specialize in smart contract audits on various blockchains and offer a range of services. They are also the leading provider of smart contract audits for decentralized finance (DeFi) projects.

300+
Audits Completed

$3B
Secured

300K
Lines of Code Audited

Contact For Audit

auditblock@gmail.com

Audit Time,
30 November 2023

Audit Report of pytbit

auditblock@gmail.com

Auditblock