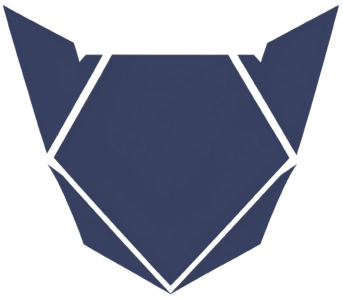


SPHINXSHIELD

Security Assessment

Bart Simpson Coin

Oct 24th, 2023

Evaluation Outcomes

Security Score

Review	Score
Overall Score	89/100
Auditor Score	83/100

Review by Section	Score
Manual Scan Score	33/57
Advance Check Score	14/19

Scoring System

This scoring system is provided to gauge the overall value of the audit. The maximum achievable score is 100, but reaching this score requires the project to meet all assessment requirements.

Our updated passing score is now set at 80 points. If a project fails to achieve at least 80% of the total score, it will result in an automatic failure.

Please refer to our notes and final assessment for more details.

Table of Contents

Summary

This audit report is tailored for **Bart Simpson Coin**, aiming to uncover potential issues and vulnerabilities within the **Bart Simpson Coin** project's source code, along with scrutinizing contract dependencies outside recognized libraries. Our audit comprises a comprehensive investigation involving Static Analysis and Manual Review techniques.

Our audit process places a strong emphasis on the following focal points:

1. Rigorous testing of smart contracts against both commonplace and rare attack vectors.
2. Evaluation of the codebase for alignment with contemporary best practices and industry standards.
3. Ensuring the contract logic is in harmony with the client's specifications and objectives.
4. A comparative analysis of the contract structure and implementation against analogous smart contracts created by industry frontrunners.
5. An exhaustive, line-by-line manual review of the entire codebase by domain experts.

The outcome of this security assessment yielded findings spanning from critical to informational. To uphold robust security standards and align with industry norms, we present the following security-driven recommendations:

1. Elevate general coding practices to optimize source code structure.
2. Implement an all-encompassing suite of unit tests to account for all conceivable use cases.
3. Enhance codebase transparency through increased commenting, particularly in externally verifiable contracts.
4. Improve clarity regarding privileged activities upon the protocol's transition to a live state.

Overview

Project Summary

Project Name	Bart Simpson Coin
Blockchain	Binance Smart Chain
Language	Solidity
Codebase	https://bscscan.com/address/0x16e79e09b3b56bcbba83667aff88dc6ca727af2e
Commit	5ba6e3e0991ad7b45fa590b3b27777bbda74ef2b6e79242a12b5a42b37b9982b

Audit Summary

Delivery Date	Oct 24th, 2023
Audit Methodology	Static Analysis, Manual Review
Key Components	$BART.sol

Vulnerability Summary

21	0	0	0	21	0
Total Findings	Resolved	Mitigated	Partially Resolved	Aknowledged	Declined

Vulnerability Level	Total	Pending	Declined	Aknowledged	Resolved
High	0	0	0	0	0
Medium	1	0	0	1	0
Low	2	0	0	2	0
Informational	18	0	0	18	0
Discussion	0	0	0	0	0

Audit Scope

ID	File	KECCAK256 or SHA256 Checksum
BRT	$BART.sol	0xec43f1a21d18fa7ac62e14796b6cbc0620a9e3ca361ae3b83d0219960676228d

Understandings

Bart Simpson Coin ($BART) is a decentralized finance (DeFi) token deployed on the Binance Smart Chain (BSC). This token offers unique features within its protocol, including static rewards for token holders and a mechanism for acquiring liquidity (LP acquisition). Below, we will delve into these features and how they operate.

Static Rewards (Reflection)

$BART employs a static reward mechanism, also known as reflection. Here's how it works:

- In every $BART transaction, two fees are imposed, totaling 10% of the transaction amount.
- The first fee (5%) is redistributed to all existing token holders. This fee operates through a rebasing mechanism.
- The second fee (5%) is accumulated internally until a sufficient amount of capital is amassed to perform an LP acquisition.

LP Acquisition

The LP (Liquidity Pool) acquisition mechanism is a critical part of the $BART token's operation. Here's how it functions:

- The LP acquisition can be indirectly triggered by any regular token transaction, as all transfers are evaluated for specific conditions that trigger the mechanism.
- The main conditions for this mechanism are whether the sender is different from the LP pair (the token's pair on PancakeSwap) and whether the accumulation threshold has been met.
- If these conditions are satisfied, the "swapAndLiquify" function is invoked using the current balance of $BART tokens held by the contract.

The "swapAndLiquify" function has several important steps:

1. It splits the contract's balance in half, accounting for potential truncation.
2. The first half is swapped into Binance Coin (BNB) through the PancakeSwap Router, which temporarily drives down the price of the $BART token.
3. The resulting BNB balance and the remaining $BART token balance are supplied to the $BART-BNB liquidity pool using the Router. The current owner of the $BART contract is designated as the recipient of the LP units.

This mechanism ensures the liquidity of the token while also facilitating reflection rewards for token holders.

Privileged Functions

The $BART contract includes several privileged functions that are restricted by the onlyOwner modifier. These functions enable modifications to contract configurations and address attributes. Some of these functions include:

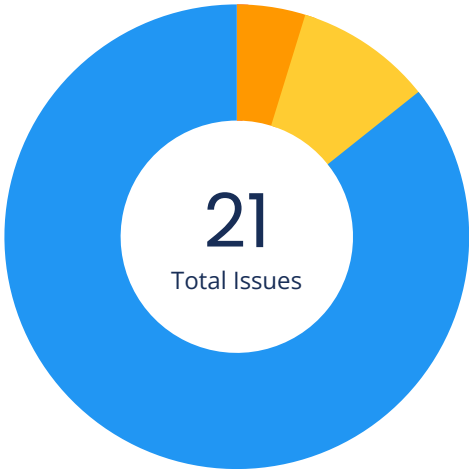
- Managing accounts for inclusion and exclusion in the fee and reward system.
- Modifying parameters such as taxation fees, liquidity fees, and maximum transaction percentages.
- Toggling the LP acquisition mechanism.

Ownership and Contract Configuration

The contract owner has privileges to configure important parameters, including tax rates, liquidity, and marketing and development wallet addresses. Additionally, the contract owner can set the maximum wallet balance and maximum buy and sell amounts for the $BART token.

This understanding provides insight into how Bart Simpson Coin operates, including its reflection-based reward system, LP acquisition mechanism, and the flexibility offered to the contract owner for managing the token's parameters and security features. It's important for users and developers to be aware of these features to participate safely and responsibly in the $BART ecosystem.

Findings



- High - 0
- Medium - 1
- Low - 2
- Informational - 18
- Discussion - 0

Location	Title	Scope	Severity	Status
$BART.sol:607	Unchecked Call Return Value	$BART	Medium	Aknowledged
$BART.sol:511,607,506,600	Use Safer Functions	$BART	Low	Aknowledged
$BART.sol:392	Clarify Return Value	$BART	Low	Aknowledged
$BART.sol:511,506,600	Prefer .call() To send()/transfer()	$BART	Informational	Aknowledged
$BART.sol:104,13,46	Recommend to Follow Code Layout Conventions	IERC20	Informational	Aknowledged
$BART.sol:581	Parameters Should Be Declared as Calldata	$BART	Informational	Aknowledged
$BART.sol:300,605,277,552,576,286,587,304,548	No Check of Address Params with Zero Address	$BART	Informational	Aknowledged
$BART.sol:124	Variables Should Be Constants	$BART	Informational	Aknowledged

Location	Title	Scope	Severity	Status
$BART.sol:487,504,509	Use Shift Operation Instead of Mul/Div	$BART	Informational	Aknowledged
$BART.sol:423	Continuous State Variable Write	$BART	Informational	Aknowledged
$BART.sol:279,409,435,378,434,272,446,410,377,216,487	Cache State Variables that are Read Multiple Times within A Function	$BART	Informational	Aknowledged
$BART.sol:411,582,288	Use ++i/--i Instead of i++/i--	$BART	Informational	Aknowledged
$BART.sol:430	Use != 0 Instead of > 0 for Unsigned Integer Comparison	$BART	Informational	Aknowledged
$BART.sol:248,300,605,219,224,228,260,233,200,210,309,592,203,277,206,256,313,243,321,304	Function Visibility Can Be External	$BART	Informational	Aknowledged
$BART.sol:11	Floating Pragma	Global	Informational	Aknowledged
$BART.sol:385,367,374,400	Optimizable Return Statement	$BART	Informational	Aknowledged
$BART.sol:599,429,436,430,287,606,69,439,432,422,421,428,237,272,60,431,250,577,261,278,442	Use CustomError Instead of String	Ownable	Informational	Aknowledged
$BART.sol:485	ReentrancyGuard Should Modify External Function	$BART	Informational	Aknowledged
$BART.sol:429,422,421,428,430,237,272,431,250,69,442	Long String in revert/require	Ownable	Informational	Aknowledged

Location	Title	Scope	Severity	Status
$BART.sol:599,491,495	Get Contract Balance of ETH in Assembly	$BART	Informational	Aknowledged
$BART.sol:429,422,421,428,69	Use Assembly to Check Zero Address	Ownable	Informational	Aknowledged

Code Security - Unchecked Call Return Value

Title	Severity	Location	Status
Unchecked Call Return Value	Medium	$BART.sol:607	Aknowledged

Description

The return value of low level calls and external calls (transfer, transferFrom and approve) should be verified since low level calls may fail and these three external function calls may only return false but not cause execution reverted once fail. If not properly handled, it might incur asset losses to users and the project party.

Code Security - Use Safer Functions

Title	Severity	Location	Status
Use Safer Functions	Low	$BART.sol:511,607,506,600	Aknowledged

Description

When calling the transfer, transferFrom, and approve functions in the ERC20 contract, there are some contracts that are not fully implemented in accordance with the ERC20 standard. In order to more comprehensively judge whether the call result meets expectations or to be compatible with different ERC20 contracts, it is recommended to use the safeTransfer, safeTransferFrom, safeApprove function to call.

Optimization Suggestion - Clarify Return Value

Title	Severity	Location	Status
Clarify Return Value	Low	$BART.sol:392	Aknowledged

Description

The returned variable is specified in the function signature, but it still calls the return statement to return a local variable defined in the function body or state variable. It is necessary to clarify whether the returned value meets expectations.

Optimization Suggestion - Prefer .call() To send()/transfer()

Title	Severity	Location	Status
Prefer .call() To send()/transfer()	Informational	$BART.sol:511,506,600	Aknowledged

Description

The send or transfer function has a limit of 2300 gas.

Optimization Suggestion - Recommend to Follow Code Layout Conventions

Title	Severity	Location	Status
Recommend to Follow Code Layout Conventions	Informational	$BART.sol:104,13,46	Aknowledged

Description

In the solidity document(https://docs.soliditylang.org/en/v0.8.17/style-guide.html), there are the following conventions for code layout: Layout contract elements in the following order: 1. Pragma statements, 2. Import statements, 3. Interfaces, 4. Libraries, 5. Contracts. Inside each contract, library or interface, use the following order: 1. Type declarations, 2. State variables, 3. Events, 4. Modifiers, 5. Functions. Functions should be grouped according to their visibility and ordered: 1. constructor, 2. receive function (if exists), 3. fallback function (if exists), 4. external, 5. public, 6. internal, 7. private.

Optimization Suggestion - Parameters Should Be Declared as Calldata

Title	Severity	Location	Status
Parameters Should Be Declared as Calldata	Informational	$BART.sol:581	Aknowledged

Description

When the compiler parses the external or public function, it can directly read the function parameters from calldata. Setting it to other storage locations may waste gas. About 300-400 gas can be saved with optimization turned off while 120-150 gas can be saved vice versa.

Optimization Suggestion - No Check of Address Params with Zero Address

Title	Severity	Location	Status
No Check of Address Params with Zero Address	Informational	$BART.sol:300,605,277,552,576,286,587,304,548	Aknowledged

Description

The input parameter of the address type in the function does not use the zero address for verification.

Optimization Suggestion - Variables Should Be Constants

Title	Severity	Location	Status
Variables Should Be Constants	Informational	$BART.sol:124	Aknowledged

Description

There are unchanging state variables in the contract, and putting unchanging state variables in storage will waste gas.

Optimization Suggestion - Use Shift Operation Instead of Mul/Div

Title	Severity	Location	Status
Use Shift Operation Instead of Mul/Div	Informational	$BART.sol:487,504,509	Aknowledged

Description

It is recommended to use shift operation instead of direct multiplication and division if possible, because shift operation is more gas-efficient.

Optimization Suggestion - Continuous State Variable Write

Title	Severity	Location	Status
Continuous State Variable Write	Informational	$BART.sol:423	Aknowledged

Description

When there are multiple continuous write operations on a state variable, the intermediate write operations are redundant and will cost more gas.

Optimization Suggestion - Cache State Variables that are Read Multiple Times within A Function

Title	Severity	Location	Status
Cache State Variables that are Read Multiple Times within A Function	Informational	$BART.sol:279,409,435,378,434,272,446,410,377,216,487	Aknowledged

Description

When a state variable is read multiple times in a function, using a local variable to cache the state variable can avoid frequently reading data from storage, thereby saving gas.

Optimization Suggestion - Use ++i/--i Instead of i++/i--

Title	Severity	Location	Status
Use ++i/--i Instead of i++/i--	Informational	$BART.sol:411,582,288	Aknowledged

Description

Compared with i++, ++i can save about 5 gas per use. Compared with i--, --i can save about 3 gas per use in for loop.

Optimization Suggestion - Use != 0 Instead of > 0 for Unsigned Integer Comparison

Title	Severity	Location	Status
Use != 0 Instead of > 0 for Unsigned Integer Comparison	Informational	$BART.sol:430	Aknowledged

Description

For unsigned integers, use !=0 for comparison, which consumes less gas than >0. When compiler optimization is turned off, about 3 gas can be saved. When compiler optimization is turned on, no gas can be saved.

Optimization Suggestion - Function Visibility Can Be External

Title	Severity	Location	Status
Function Visibility Can Be External	Informational	$BART.sol:248,300,605,219,224,228,260,233,200,210,309,592,203,277,206,256,313,243,321,304	Aknowledged

Description

Functions that are not called should be declared as external.

Optimization Suggestion - Floating Pragma

Title	Severity	Location	Status
Floating Pragma	Informational	$BART.sol:11	Aknowledged

Description

Contracts should be deployed with fixed compiler version which has been tested thoroughly or make sure to lock the contract compiler version in the project configuration. Locked compiler version ensures that contracts will not be compiled by untested compiler version.

Optimization Suggestion - Optimizable Return Statement

Title	Severity	Location	Status
Optimizable Return Statement	Informational	$BART.sol:385,367,374,400	Aknowledged

Description

The returned variable is specified in the function signature, but the return statement is still displayed in the function body, which will increase gas consumption.

Optimization Suggestion - Use CustomError Instead of String

Title	Severity	Location	Status
Use CustomError Instead of String	Informational	$BART.sol:599,429,436,430,287,606,69,439,432,422,421,428,237,272,60,431,250,577,261,278,442	Aknowledged

Description

When using require or revert, CustomError is more gas efficient than string description, as the error message described using CustomError is only compiled into four bytes. Especially when string exceeds 32 bytes, more gas will be consumed. Generally, around 250-270 gas can be saved for one CustomError replacement when compiler optimization is turned off, 60-80 gas can be saved even if compiler optimization is turned on.

Optimization Suggestion - ReentrancyGuard Should Modify External Function

Title	Severity	Location	Status
ReentrancyGuard Should Modify External Function	Informational	$BART.sol:485	Aknowledged

Description

The reentrancy guard modifier should modify the external function, because reentrancy vulnerabilities often occur in external calls.

Optimization Suggestion - Long String in revert/require

Title	Severity	Location	Status
Long String in revert/require	Informational	$BART.sol:429,422,421,428,430,237,272,431,250,69,442	Aknowledged

Description

If the string parameter in the revert/require function exceeds 32 bytes, more gas will be consumed.

Optimization Suggestion - Get Contract Balance of ETH in Assembly

Title	Severity	Location	Status
Get Contract Balance of ETH in Assembly	Informational	$BART.sol:599,491,495	Aknowledged

Description

Using the selfbalance and balance opcodes to get the ETH balance of the contract in assembly saves gas compared to getting the ETH balance through address(this).balance and xx.balance. When compiler optimization is turned off, about 210-250 gas can be saved, and when compiler optimization is turned on, about 50-100 gas can be saved.

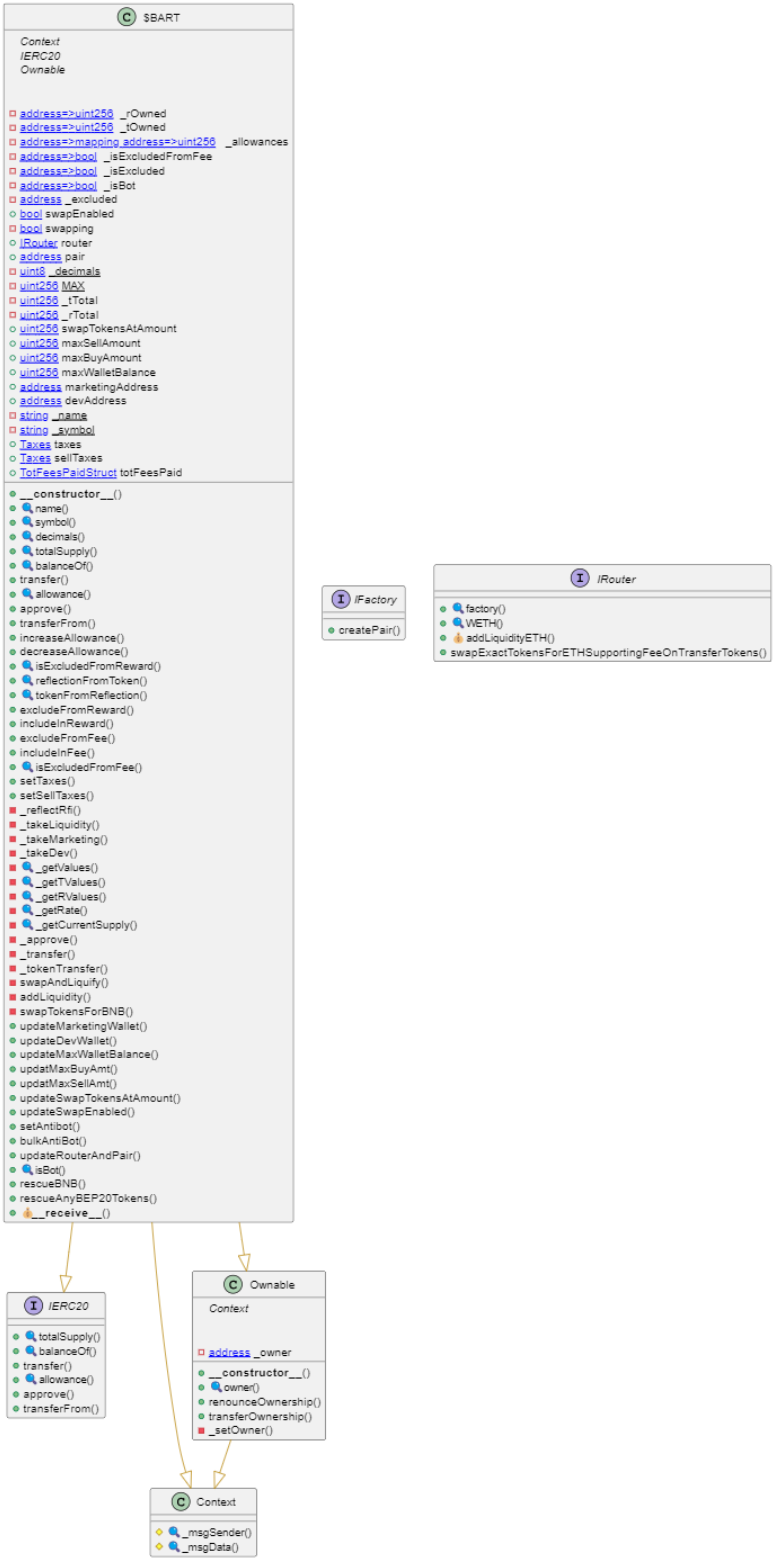
Optimization Suggestion - Use Assembly to Check Zero Address

Title	Severity	Location	Status
Use Assembly to Check Zero Address	Informational	$BART.sol:429,422,421,428,69	Aknowledged

Description

Using assembly to check zero address can save gas. About 18 gas can be saved in each call.

PlantUML


$BART
Context
IERC20
Ownable
address=>uint256 _rOwned
address=>uint256 _tOwned
address=>mapping address=>uint256 _allowances
address=>bool _isExcludedFromFee
address=>bool _isExcluded
address=>bool _isBot
address _excluded
bool swapEnabled
bool swapping
IRouter router
address pair
uint8 _decimals
uint256 MAX
uint256 _tTotal
uint256 _rTotal
uint256 swapTokensAtAmount
uint256 maxSellAmount
uint256 maxBuyAmount
uint256 maxWalletBalance
address marketingAddress
address devAddress
string _name
string _symbol
Taxes taxes
Taxes sellTaxes
TotFeesPaidStruct totFeesPaid
__constructor__()
name()
symbol()
decimals()
totalSupply()
balanceOf()
transfer()
allowance()
approve()
transferFrom()
increaseAllowance()
decreaseAllowance()
isExcludedFromReward()
reflectionFromToken()
tokenFromReflection()
excludeFromReward()
includeInReward()
excludeFromFee()
includeInFee()
isExcludedFromFee()
setTaxes()
setSellTaxes()
_reflectRfi()
_takeLiquidity()
_takeMarketing()
_takeDev()
_getValues()
_getTValues()
_getRValues()
_getRate()
_getCurrentSupply()
_approve()
_transfer()
_tokenTransfer()
swapAndLiquify()
addLiquidity()
swapTokensForBNB()
updateMarketingWallet()
updateDevWallet()
updateMaxWalletBalance()
updatMaxBuyAmt()
updatMaxSellAmt()
updateSwapTokensAtAmount()
updateSwapEnabled()
setAntibot()
bulkAntiBot()
updateRouterAndPair()
isBot()
rescueBNB()
rescueAnyBEP20Tokens()
__receive__()
IFactory
createPair()
IRouter
factory()
WETH()
addLiquidityETH()
swapExactTokensForETHSupportingFeeOnTransferTokens()
IERC20
totalSupply()
balanceOf()
transfer()
allowance()
approve()
transferFrom()
Ownable
Context
address _owner
__constructor__()
owner()
renounceOwnership()
transferOwnership()
_setOwner()
Context
_msgSender()
_msgData()

Appendix

Finding Categories

Security and Best Practices

1. Unchecked Call Return Value: Smart contracts should undergo scrutiny for unauthenticated storage access, which can lead to unauthorized data tampering.
2. Use Safer Functions: Utilize functions known for their secure design to mitigate potential security vulnerabilities. Review functions for enhanced security.
3. Clarify Return Value: Clearly define return values for functions to avoid ambiguity and potential vulnerabilities.
4. Prefer .call() To send()/transfer(): Employ .call() instead of send()/transfer() for external contract calls to minimize security risks.
5. Recommend to Follow Code Layout Conventions: Strict adherence to established code layout conventions can significantly improve code readability and maintainability.
6. Parameters Should Be Declared as Calldata: Declare function parameters as calldata when not modified within the function to enhance security and reduce gas consumption.
7. No Check of Address Params with Zero Address: Verification of address parameters should include checks to ensure that the address is not the zero address.
8. Variables Should Be Constants: Declare variables as constants if they do not change after initialization, improving security and readability.
9. Use Shift Operation Instead of Mul/Div: Opt for bitwise shift operations (<< and >>) instead of multiplication and division operations to enhance efficiency and gas savings.
10. Continuous State Variable Write: Minimize repetitive state variable writes in loops, as it can increase gas consumption.
11. Cache State Variables that are Read Multiple Times within A Function: If a state variable is read multiple times within a function, consider caching it to optimize gas usage.
12. Use ++i/--i Instead of i++/i--: Use the prefix increment and decrement operators (++i/--i) instead of postfix operators (i++/i--) for more gas-efficient code.
13. Use != 0 Instead of > 0 for Unsigned Integer Comparison: When comparing unsigned integers, use "!= 0" instead of "> 0" for improved readability and consistency.
14. Function Visibility Can Be External: Enhance gas efficiency by setting functions to external visibility if they are accessible only from within the contract.
15. Floating Pragma: Ensure that your Solidity pragma remains consistent for added contract security.
16. Optimizable Return Statement: Make return statements in functions optimizable by placing them near the end of functions to minimize gas usage.
17. Use CustomError Instead of String: Opt for custom error codes instead of string error messages for more efficient contract operation.
18. ReentrancyGuard Should Modify External Function: Ensure that a ReentrancyGuard modifies external functions rather than internal ones.
19. Long String in revert/require: Long revert or require strings can increase gas usage and should be optimized for gas efficiency.
20. Get Contract Balance of ETH in Assembly: To check the contract's ETH balance efficiently, consider using assembly.
21. Use Assembly to Check Zero Address: Optimized assembly checks can be employed to verify zero addresses efficiently.
22. Secure Project Management: Adhering to best practices in project management ensures secure and efficient development processes.
23. Code Documentation: Comprehensive code documentation is essential for team collaboration and future code maintenance.
24. Trusted Sources for External Contracts: Ensure that external contracts are sourced from reputable and verified developers.
25. Regular Code Audits: Regular code audits should be performed to identify and resolve security and functionality issues.

KECCAK256 or SHA256 Checksum Verification

Checksum verification is a critical component of smart contract development. It ensures the integrity of contract deployment and code execution by confirming that the bytecode being executed matches the intended source code. The following details the KECCAK256 and SHA256 checksum verification process.

KECCAK256 Checksum Verification:

- Checksum Definition: KECCAK256 is a cryptographic hashing function used in Ethereum to create a checksum of the contract bytecode. It is part of the Ethereum Name Service (ENS) standard.
- Use Cases: KECCAK256 checksums are used in ENS for verification of Ethereum addresses. They help prevent unintended transfers due to typos or errors.
- Checksum Process: The KECCAK256 checksum is created by taking the SHA3 hash of the lowercase hexadecimal Ethereum address, and then converting it to the corresponding checksum address by replacing characters with uppercase letters.

SHA256 Checksum Verification:

- Checksum Definition: SHA256 is a widely used cryptographic hash function, often employed to verify the integrity of data and contracts.
- Use Cases: SHA256 checksums are widely used in software development, including the verification of software downloads and smart contracts.
- Checksum Process: The SHA256 checksum is generated by applying the SHA256 hashing algorithm to the content of the contract. This results in a fixed-length hexadecimal value that is compared to the expected value to verify the contract's integrity.

Importance of Checksum Verification:

- Checksum verification ensures that smart contracts are executed as intended, preventing tampering and security vulnerabilities.
- It is a security best practice to verify that the deployed bytecode matches the intended source code, reducing the risk of unexpected behavior.

Best Practices:

- Always use checksum verification in situations where it is essential to verify Ethereum addresses or contract integrity.
- Implement checksum verification to ensure that contract deployment and interactions occur as intended.
- Verify the validity of contract deployments and the integrity of the code during development and deployment phases.

Website Scan

https://bartnetwork.com/

Network Security

High | 0 Attentions

Application Security

High | 4 Attentions

DNS Security

High | 0 Attentions

Network Security

9 Passed **0 Attention**

FTP Service Anonymous LOGIN	NO
VNC Service Accesible	NO
RDP Service Accesible	NO
LDAP Service Accesible	NO
PPTP Service Accesible	NO
RSYNC Service Accesible	NO
SSH Weak Cipher	NO
SSH Support Weak MAC	NO
CVE on the Related Service	NO

Application Security

9 Passed | 4 Attention

Missing X-Frame-Options Header	YES
Missing HSTS header	YES
Missing X-Content-Type-Options Header	YES
Missing Content Security Policy (CSP)	YES
HTTP Access Allowed	NO
Self-Signed Certificate	NO
Wrong Host Certificate	NO
Expired Certificate	NO
SSL/TLS Supports Weak Cipher	NO
Support SSL Protocols	NO
Support TLS Weak Version	NO

DNS Health

10 Passed | 0 Attention

Missing SPF Record	NO
Missing DMARC Record	NO
Missing DKIM Record	NO
Ineffective SPF Record	NO
SPF Record Contains a Softfail Without DMARC	NO
Name Servers Versions Exposed	NO
Allow Recursive Queries	NO
CNAME in NS Records	NO
MX Records IPs are Private	NO
MX Records has Invalid Chars	NO

Social Media Checks

3 Passed | 7 Failed

X (Twitter)		PASS
Facebook		FAIL
Instagram		FAIL
TikTok		FAIL
YouTube		FAIL
Twich		FAIL
Telegram		PASS
Discord		FAIL
Medium		PASS
Others		FAIL

Recommendation

To enhance project credibility and outreach, we suggest having a minimum of three active social media channels and a fully functional website.

Social Media Information Notes

Unspecified Auditor Notes

Notes from the Project Owner

Fundamental Health

KYC Status

SphinxShield KYC	NO
3rd Party KYC	NO

Project Maturity Metrics

Moderately Developed	**MEDIUM**
Token Launch Date	**2023.05.01 15:30 (UTC)**
Token Market Cap (estimate)	**$733.71K**
Token/Project Age	**176 Days**

Recommendation

We strongly recommend that the project undergo the Know Your Customer (KYC) verification process with SphinxShield to enhance transparency and build trust within the crypto community. Furthermore, we encourage the project team to reach out to us promptly to rectify any inaccuracies or discrepancies in the provided information to ensure the accuracy and reliability of their project data.

Coin Tracker Analytics

Status

CoinMarketCap	YES
CoinGecko	YES
Others	YES

Recommendation

We highly recommend that the project consider integrating with multiple coin tracking platforms to expand its visibility within the cryptocurrency ecosystem. In particular, joining prominent platforms such as CoinMarketCap and CoinGecko can significantly benefit the project by increasing its reach and credibility.

CEX Holding Analytics

Status

Not available on any centralized cryptocurrency exchanges (CEX).

Recommendation

To increase your project's visibility and liquidity, we recommend pursuing listings on centralized cryptocurrency exchanges. Here's a recommendation you can use:

We strongly advise the project team to actively pursue listings on reputable centralized cryptocurrency exchanges. Being listed on these platforms can offer numerous advantages, such as increased liquidity, exposure to a broader range of traders, and enhanced credibility within the crypto community.

To facilitate this process, we recommend the following steps:

1. Research and Identify Suitable Exchanges: Conduct thorough research to identify centralized exchanges that align with your project's goals and target audience. Consider factors such as trading volume, reputation, geographical reach, and compliance with regulatory requirements.
2. Meet Compliance Requirements: Ensure that your project is compliant with all necessary legal and regulatory requirements for listing on these exchanges. This may include Know Your Customer (KYC) verification, security audits, and legal documentation.
3. Prepare a Comprehensive Listing Proposal: Create a detailed and persuasive listing proposal for each exchange you intend to approach. This proposal should highlight the unique features and benefits of your project, as well as your commitment to compliance and security.
4. Engage in Communication: Establish open lines of communication with the exchange's listing team. Be prepared to address their questions, provide requested documentation, and work closely with their team to facilitate the listing process.
5. Marketing and Community Engagement: Promote your project within the exchange's community and among your own supporters to increase visibility and trading activity upon listing.
6. Maintain Transparency: Maintain transparency and provide regular updates to your community and potential investors about the progress of listing efforts.
7. Be Patient and Persistent: Listing processes on centralized exchanges can sometimes be lengthy. Be patient and persistent in your efforts, and consider seeking the assistance of experts or advisors with experience in exchange listings if necessary.
8.

Remember that listing on centralized exchanges can significantly impact your project's growth and market accessibility. By following these steps and maintaining a professional, compliant, and communicative approach, you can increase your chances of successfully getting listed on centralized exchanges.

Disclaimer

SphinxShield, its agents, and associates provide the information and content contained within this audit report and materials (collectively referred to as "the Services") for informational and security assessment purposes only. The Services are provided "as is" without any warranty or representation of any kind, either express or implied. SphinxShield, its agents, and associates make no warranty or undertaking, and do not represent that the Services will:

- Meet customer's specific requirements.
- Achieve any intended results.
- Be compatible or work with any other software, applications, systems, or services.
- Operate without interruption.
- Meet any performance or reliability standards.
- Be error-free or that any errors or defects can or will be corrected.

In addition, SphinxShield, its agents, and associates make no representation or warranty of any kind, express or implied, as to the accuracy, reliability, or currency of any information or content provided through the Services. SphinxShield assumes no liability or responsibility for any:

- Errors, mistakes, or inaccuracies of content and materials.
- Loss or damage of any kind incurred as a result of the use of any content.
- Personal injury or property damage, of any nature whatsoever, resulting from customer's access to or use of the Services, assessment report, or other materials.

It is imperative to recognize that the Services, including any associated assessment reports or materials, should not be considered or relied upon as any form of financial, tax, legal, regulatory, or other advice.

SphinxShield provides the Services solely to the customer and for the purposes specifically identified in this agreement. The Services, assessment reports, and accompanying materials may not be relied upon by any other person or for any purpose not explicitly stated in this agreement. Copies of these materials may not be delivered to any other person without SphinxShield's prior written consent in each instance.

Furthermore, no third party or anyone acting on behalf of a third party shall be considered a third-party beneficiary of the Services, assessment reports, and any accompanying materials. No such third party shall have any rights of contribution against SphinxShield with respect to the Services, assessment reports, and any accompanying materials.

The representations and warranties of SphinxShield contained in this agreement are solely for the benefit of the customer. Accordingly, no third party or anyone acting on behalf of a third party shall be considered a third-party beneficiary of such representations and warranties. No such third party shall have any rights of contribution against SphinxShield with respect to such representations or warranties or any matter subject to or resulting in indemnification under this agreement or otherwise.

About

SphinxShield, established in 2023, is a cybersecurity and auditing firm dedicated to fortifying blockchain and cryptocurrency security. We specialize in providing comprehensive security audits and solutions, aimed at protecting digital assets and fostering a secure investment environment.

Our accomplished team of experts possesses in-depth expertise in the blockchain space, ensuring our clients receive meticulous code audits, vulnerability assessments, and expert security advice. We employ the latest industry standards and innovative auditing techniques to reveal potential vulnerabilities, guaranteeing the protection of our clients' digital assets against emerging threats.

At SphinxShield, our unwavering mission is to promote transparency, security, and compliance with industry standards, contributing to the growth of blockchain and cryptocurrency projects. As a forward-thinking company, we remain adaptable, staying current with emerging trends and technologies to consistently enhance our services.

SphinxShield is your trusted partner for securing crypto ventures, empowering you to explore the vast potential of blockchain technology with confidence.