Arbitech Solutions

AUDIT REPORT

DINARS PAY BEP20 SMART CONTRACT

Contents

Commission

Audited Project	DINARS PAY TOKEN Smart Contract
Smart Contract	0xEAbAa9978f213EDEccC626bc538a56f0E63Dcc29
Blockchain	BEP20

Arbitech Solutions was engaged by the owners of DINARS PAY TOKEN, a BEP20 token, to conduct a comprehensive audit of their primary smart contract. The objective of this audit was to accomplish the following:

- Ensure that the smart contract functions as intended.
- Identify potential security issues with the smart contract.

This report serves as a guide for comprehending the risk exposure of the smart contract and offers recommendations to enhance its security. Throughout the audit process, we conducted a thorough examination of the smart contract's codebase and subjected it to various tests to identify potential vulnerabilities and weaknesses. Our analysis encompassed the contract's logic, flow, and overall code quality to uncover any issues that could pose a risk to the smart contract. Additionally, we conducted penetration testing to assess the contract's resilience against potential attacks in different scenarios.

The findings of our audit are presented in this report along with detailed recommendations aimed at mitigating the identified risks. It's essential to acknowledge that while our audit enhances the security posture, it does not provide an absolute guarantee of security. New vulnerabilities and attack vectors may emerge over time, emphasizing the importance of conducting regular security assessments and implementing updates as neede

Moreover, our audit report is founded on the information accessible during the audit period and does not consider any prospective modifications to the smart contract. Consequently, it is crucial to perform a fresh security assessment whenever significant changes are implemented in the contract's codebase or functionality.

In essence, the audit aims to offer transparency and assurance to the stakeholders of the smart contract, including investors and developers. It signifies that the contract's codebase has undergone a comprehensive review, potential vulnerabilities have been evaluated, and any identified risks have been addressed to the best of our knowledge and capability.

Disclaimer

This report provides a limited overview of our findings based on the analysis, following good industry practices as of the report date. It focuses on cybersecurity vulnerabilities and issues related to the framework and algorithms based on smart contracts, detailed in this report. To gain a comprehensive understanding of our analysis, it is crucial to read the full report. While we have diligently conducted our analysis and produced this report, it is important to note that you should not rely on it and cannot make claims against us based on its content or production. It is essential for you to conduct your own independent investigations before making any decisions, as elaborated in the disclaimer below.

In addition to the above statement, it is important to emphasize that this report is based on the information and materials provided to us at the time of the audit. We have not conducted a thorough investigation into the company or individuals associated with the smart contract and cannot guarantee their intentions, actions, or business practices. Our audit is solely focused on identifying potential security vulnerabilities within the smart contract and providing recommendations for remediation.

Furthermore, it should be noted that our audit does not constitute an endorsement of the smart contract or the associated project. We have not evaluated the utility or value of the smart contract and cannot provide any financial or investment advice. Our role is solely to offer a technical assessment of the security of the smart contract.

Lastly, it is essential to emphasize that security is an ongoing process requiring constant attention and maintenance. The recommendations provided in this report serve as a starting point for improving the security of the smart contract, but it is crucial to regularly review and update security measures as new threats and vulnerabilities emerge.

DISCLAIMER: 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, please cease reading this report immediately and delete any downloaded or printed copies. This report is provided for informational purposes only and on a non-reliance basis, not constituting investment advice. No one shall have the right to rely on the report or its contents. Arbitech Solutions and its affiliates owe no duty of care towards you or any other person, and Arbitech Solutions makes no warranty or representation regarding the accuracy or completeness of the report. The report is provided "as is," without any conditions, warranties, or other terms, except as set out in this disclaimer. Arbitech Solutions hereby excludes all liability and responsibility, and neither you nor any other person shall have any claim against Arbitech Solutions for any loss or damage, arising from or connected with this report and its use. The analysis of security is solely based on the smart contracts alone, with no review of applications or operations for security.

DINARS PAY TOKEN Properties

Contract name	**DINARS PAY TOKEN Smart Contract**
Contract address	**0xEAbAa9978f213EDEccC626bc538a56f0E63Dcc29**
Total supply	**100 M**
Token ticker	**$DINARS**
Decimals	**18**
Token holders	**1**
Mintable	**Yes**
Burnable	**Yes**

Contract
Functions

View

i. function name() public view returns (string memory)
ii. function symbol() public view returns (string memory)
iii. function decimals() public view returns (uint8)
iv. function totalSupply() public view override returns (uint256)
v. function balanceOf(address account) public view override returns (uint256)
vi. function owner() public view virtual returns (address);
vii. function allowance(address owner, address spender) public view override returns (uint256)

Executables

i. function approve(address spender, uint256 amount) public virtual override returns (bool)
ii. function transfer(address recipient, uint256 amount) public virtual override returns (bool)
iii. function transferFrom(address sender, address recipient, uint256 amount) public virtual override returns(bool)
iv. function increaseAllowance(address spender, uint256 addedValue) public virtual returns (bool)
v. function decreaseAllowance(address spender, uint256 subtractedValue) public virtual returns (bool)
vi. function burn(address account, uint256 value) public onlyOwner
vii. function mint(address account, uint256 value) public onlyOwner
viii. function renounceOwnership() public virtual onlyOwner

Checklist

Compiler errors.	Passed
Possible delays in data delivery.	Passed
Integer Overflow and Underflow.	Passed
Race Conditions and Reentrancy.	Passed
DoS with Revert.	Passed
DoS with Arbitech gas limit.	Passed
Economy model of the contract.	Passed
Private user data leaks.	Passed
Malicious Events Log.	Passed
Scoping and Declarations.	Passed
Uninitialized storage pointers.	Passed
Arithmetic accuracy.	Passed
Design Logic.	Passed
Impact of the exchange rate.	Passed
Oracle Calls.	Passed
Cross-function race conditions.	Passed
Fallback function security.	Passed
Front Running.	Passed
Safe Open Zeppelin contracts and implementation usage.	Passed
Whitepaper-Website-Contract correlation.	Not Checked

Owner privilege

DINARS PAY TOKEN Contract:

```
function transfer(address recipient, uint256 amount) public virtual override returns (bool) {    infinite ga
    _transfer(_msgSender(), recipient, amount);
    return true;
}
```

This function is designed to transfer a specified amount of tokens from the caller (_msgSender()) to the designated recipient. It utilizes the internal _transfer function, which presumably handles the actual transfer logic, and then returns true to indicate the success of the transfer.

```
function transferFrom(address sender, address recipient,uint256 amount    infinite gas
) public virtual override returns (bool) {
    _transfer(sender, recipient, amount);
    uint256 currentAllowance = _allowances[sender][_msgSender()];
    require(currentAllowance >= amount, "ERC20: transfer amount exceeds allowance");
    unchecked {
        _approve(sender, _msgSender(), currentAllowance - amount);
    }
    return true;
}
```

The transferFrom function is part of an ERC-20 token smart contract. It enables the controlled transfer of tokens from one address (sender) to another (recipient). Here's how it works:

The function begins by invoking an internal _transfer function, moving a specified amount of tokens from the sender to the recipient.

Next, it checks the current allowance approved by the sender for the caller (_msgSender()), ensuring it is sufficient for the intended transfer. If the allowance is inadequate, a requirement error is triggered.

Assuming the allowance is satisfactory, the function updates the allowance by decreasing it by the transferred amount. This adjustment accurately reflects the remaining approved balance after the transfer.

Finally, the function returns true to indicate the successful execution of the entire process, including the token transfer and allowance update. Overall, transferFrom ensures a secure and controlled mechanism for transferring tokens within approved limits.

```
function approve(address spender, uint256 amount) public virtual override returns (bool) {
    _approve(_msgSender(), spender, amount);
    return true;
}
```

The _approve function is an internal and virtual function within an ERC-20 token smart contract. It facilitates the setting of allowances for token transfers from an owner to a designated spender.

The function begins by checking that neither the owner nor the spender is the zero address, ensuring the validity of the approval process. Subsequently, it sets the allowance in the _allowances mapping, specifying how many tokens the spender can transfer on behalf of the owner. Finally, the function emits an Approval event, providing transparency about the approval details such as the owner, spender, and the approved amount.

```
function mint(address account, uint256 value) public onlyOwner {    infinite gas
    _mint(account, value);
}
```

This function allows the owner of the contract to create new tokens and assign them to a specific address (account). The number of tokens to be minted is specified by the value parameter. The onlyOwner modifier ensures that only the owner of the contract has the authority to invoke this minting function, thereby controlling the token issuance process.

```
function burn(address account, uint256 value) public onlyOwner {    infinite gas
    _burn(account, value);
}
```

This function allows the owner of the contract to destroy (burn) a certain amount of tokens held by a specified address (account). The number of tokens to be burned is determined by the value parameter. The onlyOwner modifier ensures that only the owner of the contract has the authority to invoke this burning function, providing exclusive control over the token removal process.

```
function _transferOwnership(address newOwner) internal virtual {    27981 gas
    address oldOwner = _owner;
    _owner = newOwner;
    emit OwnershipTransferred(oldOwner, newOwner);
}
```

The _transferOwnership function transfers ownership of the smart contract to a new address (newOwner). It updates the _owner state variable, emits an OwnershipTransferred event with details of the old and new owners.

```
function increaseAllowance(address spender, uint256 addedValue) public virtual returns (bool) {
    _approve(_msgSender(), spender, _allowances[_msgSender()][spender] + addedValue);
    return true;
}
```

This will increase approval number of tokens to spender address. "spender" is the address whose allowance will increase and "addedValue" are number of tokens which are going to be added in

```
function decreaseAllowance(address spender, uint256 subtractedValue) public virtual returns (bool)
    uint256 currentAllowance = _allowances[_msgSender()][spender];
    require(currentAllowance >= subtractedValue, "decreased allowance below zero");
    unchecked {
        _approve(_msgSender(), spender, currentAllowance - subtractedValue);
    }
    return true;
}
```

current allowance. approve should be called when _allowances[spender] == 0. To increment allowed value is better to use this function to avoid 2 calls (and wait until the first transaction is mined) From DINARS PAY Token. Sol.

Atomically decreases the allowance granted to `spender` by the caller. This is an alternative to {approve} that can be used as mitigation for problems described in {IBEP20-approve}. Emits an {Approval} eventindicating the updated allowance.

Requirements: `spender` cannot be the zero address. `spender` must have allowance for the caller of at least

`subtractedValue.

Main Category	Subcategory	Result
Contract Programmig	Solidity version not specified	Passed
	Solidity version too old	Passed
	Integer overflow/underflow	Passed
	Function input parameters lack of check	Passed
	Function input parameters check bypass	Passed
	Function access control lacks management	Passed
	Critical operation lacks event log	Passed
	Human/contract checks bypass	Passed
	Random number generation/use vulnerability	N/A
	Fallback function misuse	N/A
	Race condition	Passed
	Logical vulnerability	Passed
	Other programming issues	Passed
Code Specificatin	Visibility not explicitly declared	Passed
	Var. storage location not explicitly declared	Passed
	Use keywords/functions to be deprecated	Passed
	Other code specification issues	Passed
Business Risk	The maximum limit for mintage not set	Passed
	“Short Address” Attack	Passed
	“Double Spend” Attack	Passed

Overall Audit Result: Passed

Executive Summary

According to the standard audit assessment, Customer`s solidity smart contract is secure. Again, it is recommended to perform an Extensive audit assessment to bring a more assured conclusion.

Code Quality

The DINARS PAY BEP20 Token protocol is built on a single smart contract, which inherits other contractssuch as BEP20, BEP20Burnable. These contracts are well-written and compact, and the libraries used in the protocol are part of its logical algorithm. Once deployed on the blockchain, the smart contract is assigned a specific address and its properties and methods can be reused by other contracts in the protocol. However, the ARBITECH SOLUTIONS team has not provided scenario and unit test scripts to determine the code's integrity in an automated way. Additionally, the code lacks comments, which can provide valuable documentation for functions and return variables.

Documentation

As mentioned above, it's recommended to write comments in the smart contract code, so anyone can quickly understand the programming flow as well as complex code logic. We were given a DINARS PAY TOKEN BEP20 Token smart contract code in the form of File.

Audit Findings

Low

(1) Compiler version can be upgraded.

```
pragma solidity ^0.8.0;
```

While this doesn't present any security vulnerabilities, utilizing the latest compiler version can aid in avoiding potential compiler-level bugs.

Approve ()

```
function approve(address spender, uint256 amount) public virtual override returns (bool) {
    _approve(_msgSender(), spender, amount);
    return true;
}
```

The method enables a designated address to spend a defined amount of tokens on behalf of the message sender. Nevertheless, altering the allowance could pose a risk of someone exploiting both the old and new allowances due to transaction sequencing. To address this potential issue, it is recommended to initially reduce the spender's allowance to zero before establishing the desired value.

```
function increaseAllowance(address spender, uint256 addedValue) public virtual returns (bool)
    _approve(_msgSender(), spender, _allowances[_msgSender()][spender].add(addedValue));
    return true;
}
```

IncreaseAllowance()

The "IncreaseAllowance" function in smart contract development is utilized to augment the approved number of tokens for a specified spender address. The spender address, representing the entity with an increased allowance, receives an additional quantity of tokens specified by the "addedValue."
It is crucial to emphasize that invoking the "IncreaseAllowance" function is recommended only when the current allowance for the spender address is 0. This precautionary measure helps prevent unintended overwriting of previously approved allowances.

Conclusion

The successful audit on the BscScan Network indicates that the Smart Contract code has met certain standards. However, the low-severity warnings raise considerations, emphasizing potential vulnerabilities that may impact new holders. The audit team's suggestion to make changes for increased security is an important aspect to weigh, especially for instilling confidence in new investors.

While the decision to implement changes ultimately rests with the contract owners, it's important to recognize that addressing potential vulnerabilities can contribute to the long-term stability and trustworthiness of the contract. The audit report likely included valuable insights and recommendations for best practices, aligning with the industry's security standards.

Moreover, the absence of comments in the code poses challenges for future developers who may need to understand and modify the code. Commenting serves as a crucial documentation tool, providing clarity on the code's logic, functions, and expected inputs/outputs. It not only eases the comprehension of the code but also enhances maintainability, enabling smoother updates and adaptations as needed.

Audit report contains all found security vulnerabilities and other issues in the reviewed code.

Security state of the reviewed contract is “Secure”.

Our Methodology

We value transparency and prefer to engage in a collaborative process during our reviews. Our security audit goals focus on enhancing the quality of the systems we assess and striving for effective remediation to ensure the protection of users. The following outlines the methodology employed in our security audit process.

Manual Code Review:

To provide further detail, the manual code review process entails a meticulous and systematic examination of the smart contract code. This involves a line-by-line analysis to identify potential vulnerabilities and weaknesses. Reviewers focus on critical areas, including access controls, input validation, exception handling, and error checking.

Furthermore, the assessment includes an evaluation of the contract's adherence to industry best practices and standards, such as the BEP20 and ERC721 standards. Compliance with relevant regulatory frameworks is also checked. The contract's overall design, scalability, and efficiency are scrutinized, with identification of optimization opportunities for improved performance.

Additionally, reviewers analyze the smart contract's dependencies, encompassing third-party libraries or external APIs. This ensures that these dependencies are secure and devoid of vulnerabilities that could compromise the smart contract's security.

Vulnerability Analysis:

In addition to the earlier mentioned techniques, our audit process encompasses a thorough review of the project's documentation and commenting practices. We validate the documentation's currency and accuracy, ensuring it precisely mirrors the current state of the code. Moreover, we assess the clarity of code comments, verifying that they offer sufficient explanations for the code's functionality.

Furthermore, scrutiny is applied to the project's configuration and deployment process to guarantee adherence to best practices in secure software development. This involves verifying that the system architecture minimizes the attack surface and safeguards sensitive information. The project's continuous integration and continuous deployment (CI/CD) pipeline are also examined for secure and automated deployment practices.

Ultimately, we furnish a comprehensive report detailing our findings and recommendations to enhance the code's security and functionality. This report encompasses an overview of the project, our audit methodology, a description of identified vulnerabilities, and suggestions for mitigation.

Additionally, we provide a risk assessment and a prioritized list of recommended actions based on the severity of each vulnerability.

Documenting Results:

We employ white-box penetration testing to simulate an attacker with internal knowledge of the system. This approach is crucial as many vulnerabilities may not be evident through manual code analysis alone. In this phase, we execute diverse attacks, attempting to exploit any identified vulnerability. Each successful or unsuccessful attack, along with its corresponding countermeasures, is meticulously documented.

Subsequently, collaboration with developers ensues to remediate the identified vulnerabilities. The system is retested to confirm the effectiveness of the remediation efforts. Finally, we compile and publish a comprehensive final report encompassing a summary of our findings, recommendations, and the status of remediation.

Suggested Solutions:

To provide further clarification, once our audit is concluded and the report is delivered, the responsibility shifts to the development team and deployment engineers to review and execute our recommendations. They must prioritize which issues to address first, considering factors such as severity, impact, and the feasibility of remediation.

Immediate mitigations may be applied to diminish the risk of exploitation in live deployments while the team concurrently works on devising a permanent solution. For instance, a patch or hotfix may be released as a temporary measure to address a critical vulnerability until a more lasting fix can be implemented.

The remediation engineering process for future releases involves implementing changes to the code to rectify vulnerabilities and prevent the recurrence of similar issues. This could entail code restructuring, addition of further checks and validation, or incorporation of new security features. The development team may also need to update documentation, conduct supplementary testing, and re-audit the code to verify the effectiveness of the remediation.

Disclaimers

Privacy Arbitech Solutions Disclaimer

The Arbitech Solutions team has conducted a comprehensive analysis of the smart contract and its source code, adhering to industry best practices. The audit aimed to identify cybersecurity vulnerabilities, verify the intended functionality, and ensure successful compilation and deployment of the smart contract.

It is crucial to emphasize that the audit does not serve as an absolute guarantee of the code's security, and the report does not claim a comprehensive assessment of the contract's safety or its bug-free status. The team acknowledges the limitless nature of potential test cases, and there may be undiscovered vulnerabilities or issues.

To enhance the smart contract's security further, the team recommends the implementation of a bug bounty program. This approach involves inviting external parties to attempt to discover vulnerabilities and issues, with rewards offered to those who successfully identify such issues. Integrating bug bounty programs is a common practice in the blockchain industry, serving as a supplementary measure to formal audits and bolstering confidence in the contract's security.

Technical Disclaimer

In addition to vulnerabilities in the smart contract code, it's crucial to acknowledge that the platform and programming language used for deploying and executing the smart contract can also introduce their own vulnerabilities. For instance, the BEP20 platform is not immune to hacks, and there have been instances where smart contracts deployed on the BEP20 platform were exploited.

While the audit provides a thorough review of the smart contract code and identifies potential vulnerabilities, it cannot ensure the absolute security of the smart contract. The dynamic nature of the environment means that new vulnerabilities may emerge in the future, and existing vulnerabilities could be exploited in unforeseen ways.

Therefore, it is advisable for contract owners and investors to take proactive measures to ensure the ongoing security of the smart contract. This may involve regular security audits, implementation of best practices for secure smart contract development, and staying informed about the latest security threats and vulnerabilities in the platform and programming language used for deploying the smart contract.