Cyberscope

Audit Report

My Bro

December 2024

Repository : https://github.com/breadNbutter42/BRO/tree/main

Commit : a09a1f138f8113fcf0653c8347c43ed672b6fdc8

Audited by © cyberscope

Analysis

● Critical ● Medium ● Minor / Informative ● Pass

Severity	Code	Description	Status
●	ST	Stops Transactions	Passed
●	OTUT	Transfers User's Tokens	Passed
●	ELFM	Exceeds Fees Limit	Passed
●	MT	Mints Tokens	Passed
●	BT	Burns Tokens	Passed
●	BC	Blacklists Addresses	Passed

Diagnostics

● Critical ● Medium ● Minor / Informative

Severity	Code	Description	Status
●	MTS	Misaligned Token Supply	Unresolved
●	CO	Code Optimization	Unresolved
●	DDP	Decimal Division Precision	Unresolved
●	HV	Hardcoded Values	Unresolved
●	IDI	Immutable Declaration Improvement	Unresolved
●	IOV	Inconsistent Openzeppelin Versioning	Unresolved
●	MCF	Missing Claim Functionality	Unresolved
●	MEE	Missing Events Emission	Unresolved
●	PTRP	Potential Transfer Revert Propagation	Unresolved
●	RC	Repetitive Calculations	Unresolved
●	RCS	Redundant Conditional Statements	Unresolved
●	L04	Conformance to Solidity Naming Conventions	Unresolved

Table of Contents

Risk Classification

The criticality of findings in Cyberscope’s smart contract audits is determined by evaluating multiple variables. The two primary variables are:

1. **Likelihood of Exploitation**: This considers how easily an attack can be executed, including the economic feasibility for an attacker.
2. **Impact of Exploitation**: This assesses the potential consequences of an attack, particularly in terms of the loss of funds or disruption to the contract's functionality.

Based on these variables, findings are categorized into the following severity levels:

1. **Critical**: Indicates a vulnerability that is both highly likely to be exploited and can result in significant fund loss or severe disruption. Immediate action is required to address these issues.
2. **Medium**: Refers to vulnerabilities that are either less likely to be exploited or would have a moderate impact if exploited. These issues should be addressed in due course to ensure overall contract security.
3. **Minor**: Involves vulnerabilities that are unlikely to be exploited and would have a minor impact. These findings should still be considered for resolution to maintain best practices in security.
4. **Informative**: Points out potential improvements or informational notes that do not pose an immediate risk. Addressing these can enhance the overall quality and robustness of the contract.

Severity	Likelihood / Impact of Exploitation
● Critical	Highly Likely / High Impact
● Medium	Less Likely / High Impact or Highly Likely/ Lower Impact
● Minor / Informative	Unlikely / Low to no Impact

Review

Repository	https://github.com/breadNbutter42/BRO/tree/main
Commit	a09a1f138f8113fcf0653c8347c43ed672b6fdc8

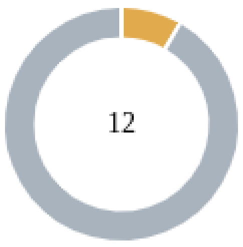
Audit Updates

Initial Audit	10 Dec 2024
Test Deploy	https://sepolia.etherscan.io/address/0x2dCd5C5523FB890f20C4414F59B6D968bfFAFBed

Source Files

Filename	SHA256
BroTokenWithPresale.sol	0cb7e1e5ea04188c2dcb9f2c8061b63a334b24f07d4f9e4b4ded4c63f0fc2548

Findings Breakdown



●	Critical	0
●	Medium	1
●	Minor / Informative	11

Severity		Unresolved	Acknowledged	Resolved	Other
●	Critical	0	0	0	0
●	Medium	1	0	0	0
●	Minor / Informative	11	0	0	0

MTS - Misaligned Token Supply

Criticality	Medium
Location	BroTokenWithPresale.sol#L174
Status	Unresolved

Description

The `decimals` field of an ERC20 token contract specifies the number of decimal places the token uses. For example, if `decimals` is set to 8, the smallest unit of the token is 0.00000001, and if set to 18, the smallest unit is 0.000000000000000001.

In this case, the token defines a total supply of 420,690,000,000,000 with a `decimals` count of 18. This means that a total of 0.00042069 tokens will be minted. This low token count could confuse users and potentially misalign with the intended supply.

```
super._update(address(0), address(this), TOTAL_SUPPLY_WEI); //Mint
total supply to this contract, to later make LP and presale airdrop
```

Recommendation

Minting a number of tokens that aligns with the decimal precision follows best practices and ensures consistency and optimal performance for users.

CO - Code Optimization

Criticality	Minor / Informative
Location	BroTokenWithPresale.sol#L259
Status	Unresolved

Description

The contract implements the `emergencyWithdrawWithFee` function, allowing users to withdraw their funds before the presale finalizes. When a user withdraws their funds, the slot in the array holding the data is overwritten with a zero value instead of being deleted. This method overlooks the potential for gas optimization by not deleting the element from the array and all relevant information about the user, including updating the `presaleBuyers` array.

```
function emergencyWithdrawWithFee() external nonReentrant {
...
totalAvaxUserSent[msg.sender] = 0; //Reset user's total AVAX sent to 0
 ...
}
```

Recommendation

The team is advised to take these segments into consideration and rewrite them so the runtime will be more performant. That way it will improve the efficiency and performance of the source code and reduce the cost of executing it.

DDP - Decimal Division Precision

Criticality	Minor / Informative
Location	BroTokenWithPresale.sol#L355
Status	Unresolved

Description

Division of decimal (fixed point) numbers can result in rounding errors due to the way that division is implemented in Solidity. Thus, it may produce issues with precise calculations with decimal numbers.

Solidity represents decimal numbers as integers, with the decimal point implied by the number of decimal places specified in the type (e.g. decimal with 18 decimal places). When a division is performed with decimal numbers, the result is also represented as an integer, with the decimal point implied by the number of decimal places in the type. This can lead to rounding errors, as the result may not be able to be accurately represented as an integer with the specified number of decimal places.

Hence, the splitted shares will not have the exact precision and some funds may not be calculated as expected.

```
amount_ = (totalAvaxUserSent[buyer_] * PRESALERS_BRO_SUPPLY_WEI) /
totalAvaxPresaleWei;
```

Recommendation

The team is advised to take into consideration the rounding results that are produced from the solidity calculations. The contract could calculate the subtraction of the divided funds in the last calculation in order to avoid the division rounding issue.

HV - Hardcoded Values

Criticality	Minor / Informative
Location	BroTokenWithPresale.sol#L94
Status	Unresolved

Description

The contract contains multiple instances where numeric values are directly hardcoded into the code logic rather than being assigned to constant variables with descriptive names. Hardcoding such values can lead to several issues, including reduced code readability, increased risk of errors during updates or maintenance, and difficulty in consistently managing values throughout the contract. Hardcoded values can obscure the intent behind the numbers, making it challenging for developers to modify or for users to understand the contract effectively.

```
uint256 public constant SECONDS_FOR_WL = 5 minutes; //Seconds per
each phase, for example 5 minutes is 300 seconds
    uint256 public constant TOTAL_SUPPLY_WEI = 420690000000000;
//BRO supply 420.69T in wei
    uint256 public constant PRESALERS_BRO_SUPPLY_WEI =
(TOTAL_SUPPLY_WEI * 50) / 100; //50% of BRO supply is for the
presale buyers
    uint256 public constant LP_BRO_SUPPLY_WEI = TOTAL_SUPPLY_WEI -
PRESALERS_BRO_SUPPLY_WEI; //Remaining BRO is for automated LP
    uint256 public constant IDO_START_TIME = 1734254100;
//Whitelist phase start time in unix timestamp : Sun Dec 15 2024
04:15:00 AM GMT-0500 (Eastern Standard Time)
    uint256 public constant PRESALE_END_TIME = IDO_START_TIME - 120
minutes; //LP seeding start time in unix timestamp, must be before
IDO_START_TIME
    uint256 public constant AIRDROP_TIME = IDO_START_TIME - 119
minutes; //Date presale tokens dispersal starts, leave a buffer
since miners can vary timestamp slightly
    uint256 public constant MINIMUM_BUY_WEI = 1000000000000000000;
//1 AVAX in wei minimum buy, to prevent micro buy spam attack
hurting the airdrop phase
    uint256 public constant TOTAL_PHASES = 4;
```

Recommendation

It is recommended to replace hardcoded numeric values with variables that have meaningful names. This practice improves code readability and maintainability by clearly indicating the purpose of each value, reducing the likelihood of errors during future modifications. Additionally, consider using constant variables which provide a reliable way to centralize and manage values, improving gas optimization throughout the contract.

IDI - Immutable Declaration Improvement

Criticality	Minor / Informative
Location	BroTokenWithPresale.sol#L172
Status	Unresolved

Description

The contract declares state variables that their value is initialized once in the constructor and are not modified afterwards. The `immutable` is a special declaration for this kind of state variables that saves gas when it is defined.

```
lfjV1PairAddress
```

Recommendation

By declaring a variable as immutable, the Solidity compiler is able to make certain optimizations. This can reduce the amount of storage and computation required by the contract, and make it more gas-efficient.

IOV - Inconsistent Openzeppelin Versioning

Criticality	Minor / Informative
Location	BroTokenWithPresale.sol#L87
Status	Unresolved

Description

The smart contract utilizes the OpenZeppelin library to import key components such as the `ERC20` and `ReentrancyGuard` extensions. The contract is built using the latest v5 versions of OpenZeppelin; however, it imports components from the previous v4 versions. Specifically, the path specified for the `ReentrancyGuard` contract corresponds to the v4 implementation rather than v5. This inconsistency in using a mixed OpenZeppelin version could lead to issues such as compilation errors.

```
import "@openzeppelin/contracts/token/ERC20/ERC20.sol";
import
"@openzeppelin/contracts/token/ERC20/extensions/ERC20Permit.sol";
import "@openzeppelin/contracts/security/ReentrancyGuard.sol";
```

Recommendation

Using a stable OpenZeppelin version ensures optimal functionality and consistency in the code, enhancing runtime performance and increasing user trust.

MCF - Missing Claim Functionality

Criticality	Minor / Informative
Location	BroTokenWithPresale.sol#L244
Status	Unresolved

Description

The contract implements a presale mechanism but does not include a function for users to claim tokens themselves after the presale ends. The only way for users to claim their tokens is by waiting for their index to be processed through the `airdropBuyers` function. This discrepancy could lead to confusion among users and loss of trust.

```
function airdropBuyers(uint256 maxTransfers_) external nonReentrant {
//Airdrop function where users can set max transfers per tx
require(lpSeeded, "LP must be seeded before airdrop can start");
require(!airdropCompleted, "Airdrop has already been completed");
require(block.timestamp >= AIRDROP_TIME, "It is not yet time to airdrop the
presale buyer's tokens");
_airdrop(maxTransfers_);
}
```

Recommendation

Implementing a mechanism that allows users to claim their funds independently with minimal gas consumption will promote trust in the system and ensure overall stability.

MEE - Missing Events Emission

Criticality	Minor / Informative
Location	BroTokenWithPresale.sol#L246
Status	Unresolved

Description

The contract performs actions and state mutations from external methods that do not result in the emission of events. Emitting events for significant actions is important as it allows external parties, such as wallets or dApps, to track and monitor the activity on the contract. Without these events, it may be difficult for external parties to accurately determine the current state of the contract.

```
function airdropBuyers(uint256 maxTransfers_) external nonReentrant
{ //Airdrop function where users can set max transfers per tx
require(lpSeeded, "LP must be seeded before airdrop can start");
require(!airdropCompleted, "Airdrop has already been completed");
require(block.timestamp >= AIRDROP_TIME, "It is not yet time to
airdrop the presale buyer's tokens");
_airdrop(maxTransfers_);
}
```

Recommendation

It is recommended to include events in the code that are triggered each time a significant action is taking place within the contract. These events should include relevant details such as the user's address and the nature of the action taken. By doing so, the contract will be more transparent and easily auditable by external parties. It will also help prevent potential issues or disputes that may arise in the future.

PTRP - Potential Transfer Revert Propagation

Criticality	Minor / Informative
Location	BroTokenWithPresale.sol#L269
Status	Unresolved

Description

The contract sends funds to a `FEE_WALLET` as part of the transfer flow. This address can either be a wallet address or a contract. If the address belongs to a contract then it may revert from incoming payment. As a result, the error will propagate to the token's contract and revert the transfer.

```
payable(FEE_WALLET).transfer(fee_);
```

Recommendation

The contract should tolerate the potential revert from the underlying contracts when the interaction is part of the main transfer flow. This could be achieved by not allowing set contract addresses or by sending the funds in a non-revertable way.

RC - Repetitive Calculations

Criticality	Minor / Informative
Location	BroTokenWithPresale.sol#L277,352
Status	Unresolved

Description

The contract contains methods with multiple occurrences of the same calculation being performed. The calculation is repeated without utilizing a variable to store its result, which leads to redundant code, hinders code readability, and increases gas consumption. Each repetition of the calculation requires computational resources and can impact the performance of the contract, especially if the calculation is resource-intensive.

```
amount_ = (totalAvaxUserSent[buyer_] * PRESALERS_BRO_SUPPLY_WEI) /
totalAvaxPresaleWei;
```

Recommendation

To address this finding and enhance the efficiency and maintainability of the contract, it is recommended to refactor the code by assigning the calculation result to a variable once and then utilizing that variable throughout the method. By storing the calculation result in a variable, the contract eliminates the need for redundant calculations and optimizes code execution.

Refactoring the code to assign the calculation result to a variable has several benefits. It improves code readability by making the purpose and intent of the calculation explicit. It also reduces code redundancy, making the method more concise, easier to maintain, and gas effective. Additionally, by performing the calculation once and reusing the variable, the contract improves performance by avoiding unnecessary computations.

RCS - Redundant Conditional Statements

Criticality	Minor / Informative
Location	BroTokenWithPresale.sol#L181
Status	Unresolved

Description

The contract contains redundant conditional statements that can be simplified to improve code efficiency and performance. Conditional statements that merely return the result of an expression are unnecessary and lead to larger code size, increased memory usage, and slower execution times. By directly returning the result of the expression, the code can be made more concise and efficient, reducing gas costs and improving runtime performance. Such redundancies are common when simple comparisons or checks are performed within conditional statements, leading to redundant operations.

```
function tradingActive() public view returns (bool) { //Check if
IDO_START_TIME happened yet to open trading
if (lpSeeded) {
return block.timestamp >= IDO_START_TIME; //Return true if IDO has started
} else {
return false;
}
}
```

Recommendation

It is recommended to refactor conditional statements that return results by eliminating unnecessary code structures and directly returning the outcome of the expression. This practice minimizes the number of operations required, reduces the code footprint, and optimizes memory and gas usage. Simplifying such statements makes the code more readable and improves its overall performance.

L04 - Conformance to Solidity Naming Conventions

Criticality	Minor / Informative
Location	BroTokenWithPresale.sol
Status	Unresolved

Description

The Solidity style guide is a set of guidelines for writing clean and consistent Solidity code. Adhering to a style guide can help improve the readability and maintainability of the Solidity code, making it easier for others to understand and work with.

The followings are a few key points from the Solidity style guide:

1. Use camelCase for function and variable names, with the first letter in lowercase (e.g., myVariable, updateCounter).
2. Use PascalCase for contract, struct, and enum names, with the first letter in uppercase (e.g., MyContract, UserStruct, ErrorEnum).
3. Use uppercase for constant variables and enums (e.g., MAX_VALUE, ERROR_CODE).
4. Use indentation to improve readability and structure.
5. Use spaces between operators and after commas.
6. Use comments to explain the purpose and behavior of the code.
7. Keep lines short (around 120 characters) to improve readability.

```
*/

// SPDX-License-Identifier: MIT
// Compatible with OpenZep
uint256 public constant PRESALE_END_TIME = IDO_START_TIME -

...
```

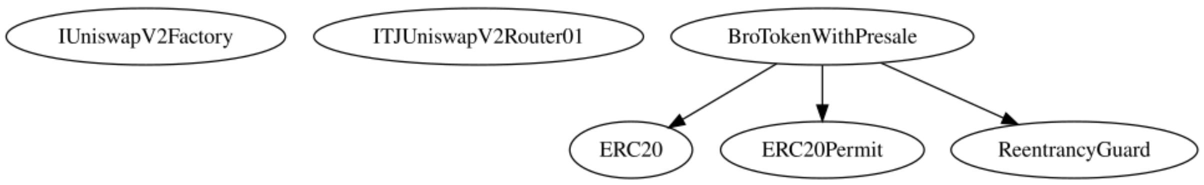
Recommendation

By following the Solidity naming convention guidelines, the codebase increased the readability, maintainability, and makes it easier to work with.
Find more information on the Solidity documentation
https://docs.soliditylang.org/en/stable/style-guide.html#naming-conventions.

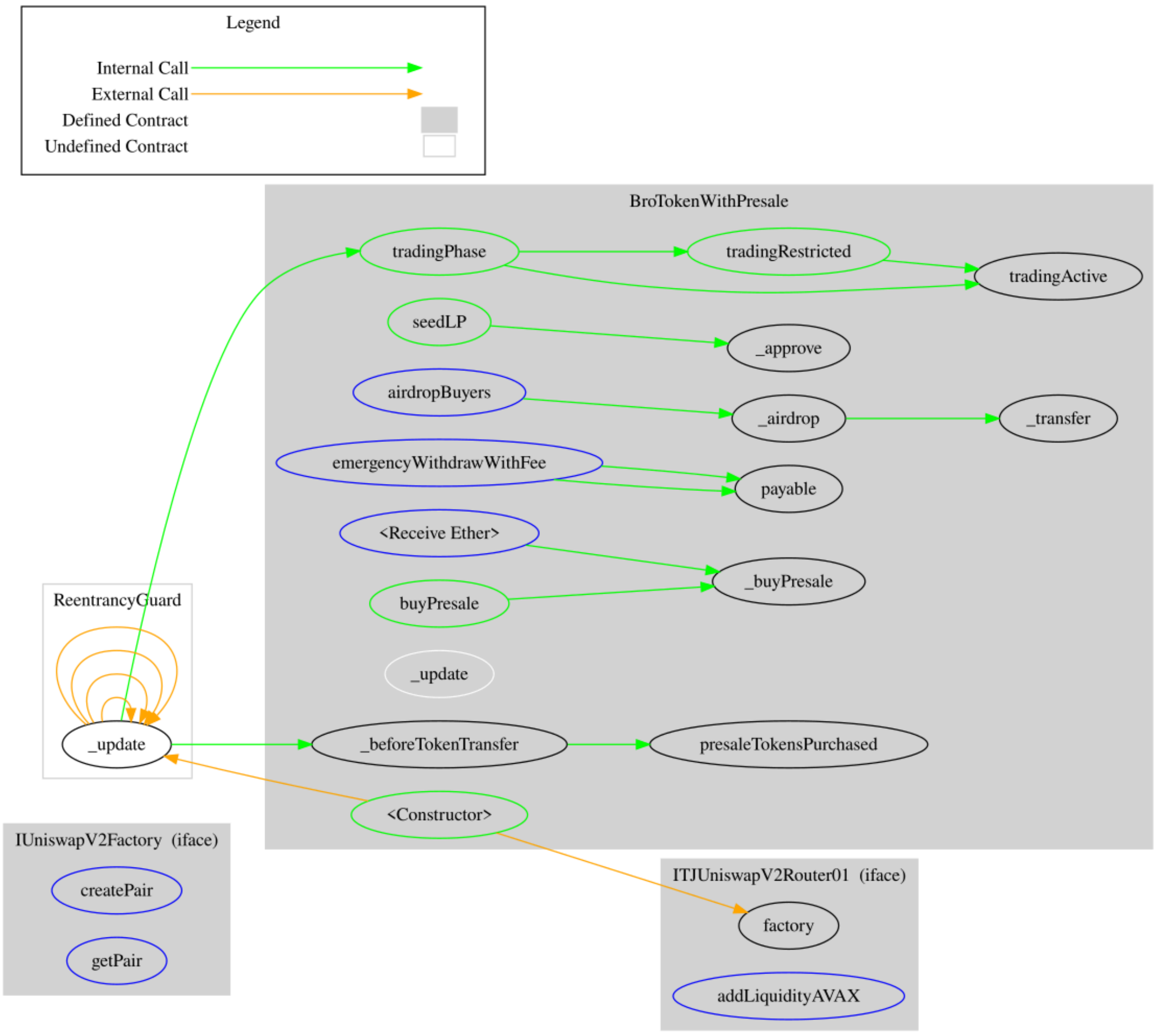
Functions Analysis

Contract	Type	Bases		
	Function Name	**Visibility**	**Mutability**	**Modifiers**
BroTokenWithPresale	Implementation	ERC20, ERC20Permit, ReentrancyGuard		
		Public	✓	ERC20 ERC20Permit
	tradingActive	Public		-
	tradingRestricted	Public		-
	tradingPhase	Public		-
	seedLP	Public	✓	nonReentrant
	airdropBuyers	External	✓	nonReentrant
	buyPresale	Public	Payable	-
	emergencyWithdrawWithFee	External	✓	nonReentrant
	presaleTokensPurchased	Public		-
	_update	Internal	✓	
	_beforeTokenTransfer	Private	✓	
	_buyPresale	Private	✓	nonReentrant
	_airdrop	Private	✓	
		External	Payable	-

Inheritance Graph



Flow Graph


Legend
Internal Call
External Call
Defined Contract
Undefined Contract
BroTokenWithPresale
tradingPhase
tradingRestricted
tradingActive
seedLP
_approve
airdropBuyers
_airdrop
_transfer
emergencyWithdrawWithFee
payable
<Receive Ether>
_buyPresale
buyPresale
_update
_beforeTokenTransfer
presaleTokensPurchased
<Constructor>
ReentrancyGuard
_update
IUniswapV2Factory (iface)
createPair
getPair
ITJUniswapV2Router01 (iface)
factory
addLiquidityAVAX

Summary

My Bro contract implements a token and a presale mechanism. This audit investigates security issues, business logic concerns and potential improvements. My Bro is an interesting project that has a friendly and growing community. The Smart Contract analysis reported no compiler error or critical issues.

Disclaimer

About Cyberscope

Cyberscope is a blockchain cybersecurity company that was founded with the vision to make web3.0 a safer place for investors and developers. Since its launch, it has worked with thousands of projects and is estimated to have secured tens of millions of investors' funds.

Cyberscope is one of the leading smart contract audit firms in the crypto space and has built a high-profile network of clients and partners.

The Cyberscope team

cyberscope.io