Cyberscope

Audit Report

buidl

February 2025

Network	BSC
Address	0x4A67A307E6c6F35117ecfDD4A2767095EB4c2B4E
Audited by	© cyberscope

Analysis

● Critical ● Medium ● Minor / Informative ● Pass

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

Diagnostics

● Critical ● Medium ● Minor / Informative

Severity	Code	Description	Status
●	PTRP	Potential Transfer Revert Propagation	Unresolved
●	CR	Code Repetition	Unresolved
●	RRA	Redundant Repeated Approvals	Unresolved
●	PLPI	Potential Liquidity Provision Inadequacy	Unresolved
●	PVC	Price Volatility Concern	Unresolved
●	RC	Repetitive Calculations	Unresolved
●	TUU	Time Units Usage	Unresolved
●	RMV	Rebase Mechanism Vulnerability	Unresolved
●	RPD	Rebase Price Distortion	Unresolved
●	PMLPR	Potential Main Liquidity Pair Removal	Unresolved
●	TSPL	Total Supply Precision Loss	Unresolved
●	RSML	Redundant SafeMath Library	Unresolved
●	IDI	Immutable Declaration Improvement	Unresolved
●	L02	State Variables could be Declared Constant	Unresolved

●	L04	Conformance to Solidity Naming Conventions	Unresolved
●	L11	Unnecessary Boolean equality	Unresolved
●	L13	Divide before Multiply Operation	Unresolved
●	L16	Validate Variable Setters	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

Contract Name	BUIDL
Compiler Version	v0.8.20+commit.a1b79de6
Optimization	200 runs
Explorer	https://bscscan.com/address/0x4a67a307e6c6f35117ecfdd4a2767095eb4c2b4e
Address	0x4a67a307e6c6f35117ecfdd4a2767095eb4c2b4e
Network	BSC
Symbol	BUIDL
Decimals	18
Total Supply	21.324,376
Badge Eligibility	Must Fix Criticals

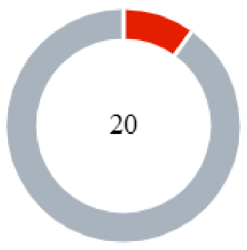
Audit Updates

Initial Audit	08 Feb 2025 https://github.com/cyberscope-io/audits/blob/main/2-buidl/v1/audit.pdf
Corrected Phase 2	18 Feb 2025

Source Files

Filename	SHA256
contracts/BUIDL.sol	c37c96fc489a72301a5b9dcbca356ed9c2836e955c0524b5fe0802d2ee05a54c

Findings Breakdown



🟥	Critical	2
🟨	Medium	0
⬜	Minor / Informative	18

Severity	Unresolved	Acknowledged	Resolved	Other
● Critical	2	0	0	0
● Medium	0	0	0	0
● Minor / Informative	18	0	0	0

ST - Stops Transactions

Criticality	Critical
Location	contracts/BUIDL.sol#L289,420
Status	Unresolved

Description

The contract owner has the authority to stop the sales for all users excluding the owner. The owner may take advantage of it by setting the `saleCooldownPeriod` to one per day. As a result, the contract may operate as a honeypot.

```
require(block.timestamp >=
mostRecentSaleTime[sender].add(saleCooldownPeriod), "Sale cooldown period
not completed yet");
```

Additionally the contract owner can set `insuredFundWallet` to a contract address and stop all the transfers as described in the `PTRP` section.

```
payable(insuredFundWallet).transfer(insuredFundPart);
```

Recommendation

The contract could embody a check for not allowing setting the `saleCooldownPeriod` more than a reasonable amount. Additionally, the team is advised to follow the recommendations outlined in the `PTPR` findings and implement the necessary steps to mitigate the identified risks. The team should carefully manage the private keys of the owner's account. We strongly recommend a powerful security mechanism that will prevent a single user from accessing the contract admin functions.

Temporary Solutions:

These measurements do not decrease the severity of the finding

- Introduce a time-locker mechanism with a reasonable delay.
- Introduce a multi-signature wallet so that many addresses will confirm the action.
- Introduce a governance model where users will vote about the actions.

Permanent Solution:

- Renouncing the ownership, which will eliminate the threats but it is non-reversible.

ELFM - Exceeds Fees Limit

Criticality	Critical
Location	contracts/BUIDL.sol#L74,354
Status	Unresolved

Description

The contract enforces a flat transfer fee of 95%, as defined in the liquidityFee array. Specifically, the third index (liquidityFee[2] = 95) is used to calculate the fee during transfers. This fee significantly exceeds the commonly accepted maximum limit of 25%, making the contract potentially non-compliant with regulatory standards and unreasonably restrictive for users. Such a high fee can discourage usage while raising concerns about malicious intent.

```
uint256[3] public liquidityFee = [5, 5, 95];
...
uint256 newLiquidityFee = gonAmount.mul(liquidityFee[2]).div(DIVIDER);
```

Recommendation

The team is advised to reduce the transfer fee to comply with the allowed maximum limit of 25%. The team could modify the liquidityFee structure to ensure that excessive fees cannot be set, either by introducing a setter function with an upper bound or by hardcoding a reasonable limit in the contract logic.

PTRP - Potential Transfer Revert Propagation

Criticality	Minor / Informative
Location	contracts/BUIDL.sol#L420
Status	Unresolved

Description

The contract sends funds to a `insuredFundWallet` 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.

In the current configuration of the contract the `changeFeeWallets` function checks if an address is a contract and reverts if it is, however contract addresses can be predetermined. A user may first determine the address of the contract, set it as any of the addresses mentioned above and then create the contract.

```
payable(insuredFundWallet).transfer(insuredFundPart);
```

Recommendation

The contract should tolerate the potential revert from the underlying contracts when the interaction is part of the main transfer flow. by sending the funds in a non-revertable way. The contract should also tolerate the potential drain of gas by using methods that send a specific amount.

CR - Code Repetition

Criticality	Minor / Informative
Location	contracts/BUIDL.sol#L227,236
Status	Unresolved

Description

The contract contains repetitive code segments. There are potential issues that can arise when using code segments in Solidity. Some of them can lead to issues like gas efficiency, complexity, readability, security, and maintainability of the source code. It is generally a good idea to try to minimize code repetition where possible.

```
if(address(token0) == address(this))
{
   uint256 toSkim = balanceOf(liquidityPairs[i]).sub(reserve0);
   if(toSkim > 0)
   {
       _basicTransfer(liquidityPairs[i], address(BURN), toSkim);
       emit LogSkim(block.timestamp, toSkim);
   }
}
else if(address(token1) == address(this))
{
   uint256 toSkim = balanceOf(liquidityPairs[i]).sub(reserve1);
   if(toSkim > 0)
   {
       _basicTransfer(liquidityPairs[i], address(BURN), toSkim);
      emit LogSkim(block.timestamp, toSkim);
   }
}
```

Recommendation

The team is advised to avoid repeating the same code in multiple places, which can make the contract easier to read and maintain. The authors could try to reuse code wherever possible, as this can help reduce the complexity and size of the contract. For instance, the contract could reuse the common code segments in an internal function in order to avoid repeating the same code in multiple places.

RRA - Redundant Repeated Approvals

Criticality	Minor / Informative
Location	contracts/BUIDL.sol#L427,443
Status	Unresolved

Description

The contract is designed to approve token transfers during the contract's operation by setting the amount to the `allowedFragments` before specific operations. This approach results in additional gas costs since the approval process is repeated for every operation execution, leading to inefficiencies and increased transaction expenses.

```
allowedFragments[address(this)][address(router)] = tokenAmount;
router.swapExactTokensForETHSupportingFeeOnTransferTokens(
    tokenAmount,
    0,
    path,
    address(this),
    block.timestamp
);
```

Recommendation

Since the approved address is a trusted third-party source, it is recommended to optimize the contract by approving the maximum amount of tokens once in the initial set of the variable, rather than before each operation. This change will reduce the overall gas consumption and improve the efficiency of the contract.

PLPI - Potential Liquidity Provision Inadequacy

Criticality	Minor / Informative
Location	contracts/BUIDL.sol#L444
Status	Unresolved

Description

The contract operates under the assumption that liquidity is consistently provided to the pair between the contract's token and the native currency. However, there is a possibility that liquidity is provided to a different pair. This inadequacy in liquidity provision in the main pair could expose the contract to risks. Specifically, during eligible transactions, where the contract attempts to swap tokens with the main pair, a failure may occur if liquidity has been added to a pair other than the primary one. Consequently, transactions triggering the swap functionality will result in a revert.

```
router.swapExactTokensForETHSupportingFeeOnTransferTokens(
    tokenAmount,
    0,
    path,
    address(this),
    block.timestamp
);
```

Recommendation

The team is advised to implement a runtime mechanism to check if the pair has adequate liquidity provisions. This feature allows the contract to omit token swaps if the pair does not have adequate liquidity provisions, significantly minimizing the risk of potential failures.

Furthermore, the team could ensure the contract has the capability to switch its active pair in case liquidity is added to another pair.

Additionally, the contract could be designed to tolerate potential reverts from the swap functionality, especially when it is a part of the main transfer flow. This can be achieved by executing the contract's token swaps in a non-reversible manner, thereby ensuring a more resilient and predictable operation.

PVC - Price Volatility Concern

Criticality	Minor / Informative
Location	contracts/BUIDL.sol#L496
Status	Unresolved

Description

The contract accumulates tokens from the taxes to swap them for ETH. The variable `gonSwapThreshold` sets a threshold where the contract will trigger the swap functionality. If the variable is set to a big number, then the contract will swap a huge amount of tokens for ETH.

It is important to note that the price of the token representing it, can be highly volatile. This means that the value of a price volatility swap involving Ether could fluctuate significantly at the triggered point, potentially leading to significant price volatility for the parties involved.

```
function updateSwapThreshold(uint256 newThreshold) external onlyOwner {
    require(newThreshold > 0, "Threshold must be greater than 0");

    gonSwapThreshold = newThreshold.mul(gonsPerFragment);
    emit SwapThresholdUpdated(newThreshold);
}
```

Recommendation

The contract could ensure that it will not sell more than a reasonable amount of tokens in a single transaction. A suggested implementation could check that the maximum amount should be less than a fixed percentage of the exchange reserves. Hence, the contract will guarantee that it cannot accumulate a huge amount of tokens in order to sell them.

RC - Repetitive Calculations

Criticality	Minor / Informative
Location	contracts/BUIDL.sol#L406
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.

```
liqudityToken.mul(gonsPerFragment)
```

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

TUU - Time Units Usage

Criticality	Minor / Informative
Location	contracts/BUIDL.sol#L84,504
Status	Unresolved

Description

The contract is using arbitrary numbers to form time-related values. As a result, it decreases the readability of the codebase and prevents the compiler to optimize the source code.

```
uint256 public saleCooldownPeriod = 86400;
require(86400 >= newCooldownPeriod, "New cooldown period can't be more
than one day");
```

Recommendation

It is good practice to use the time units reserved keywords like `seconds` , `minutes` , `hours` , `days` and `weeks` to process time-related calculations.

It's important to note that these time units are simply a shorthand notation for representing time in seconds, and do not have any effect on the actual passage of time or the execution of the contract. The time units are simply a convenience for expressing time in a more human-readable form.

RMV - Rebase Mechanism Vulnerability

Criticality	Minor / Informative
Location	contracts/BUIDL.sol#L360
Status	Unresolved

Description

The rebase mechanism dynamically adjusts the total supply and individual token balances, which can cause discrepancies in contracts that interact with the rebasing token. Specifically, decentralized applications (DApps) or liquidity pools that store token balances or total supply in state variables may experience inconsistencies after a rebase event. This can lead to issues such as incorrect price calculations, failed transactions, or vulnerabilities that allow arbitrage exploitation. Additionally, liquidity pools may have unexpected token burns or redistributions, which can affect trading mechanics.

```
function rebase() private {
    uint256 deltaTime = block.timestamp.sub(lastRebase);
    uint256 times = deltaTime.div(REBASE_INTERVAL);
    uint256 maxTimes = times > 12 ? 12 : times;

    for (uint256 i = 0; i < maxTimes; i++) {
        uint256 supplyDelta =
supply.mul(REBASE_RATE).div(REBASE_RATE_DIVIDER);
        if (supply.add(supplyDelta) >= MAX_SUPPLY)
        {
            supply = MAX_SUPPLY;
            break;
        }
        else
        {
            supply = supply.add(supplyDelta);
        }
        lastRebase = lastRebase.add(REBASE_INTERVAL);
    }
    gonsPerFragment = TOTAL_GONS.div(supply);
    emit LogRebase(block.timestamp, supply);
    manualSkim();
}
```

Recommendation

To mitigate the risks associated with the rebase mechanism, the contract owner should manually track the liquidity pairs where the token is listed. These liquidity pairs should be added to the contract's designated list of liquidity pairs to ensure proper tracking and interaction during rebase events.

Additionally, the contract should be configured to maintain compatibility with various decentralized applications (DApps), including V3 liquidity pools, token lockers, and other DeFi protocols. Implementing a mechanism to detect and adjust interactions with these external systems will help prevent inconsistencies, incorrect price calculations, or potential arbitrage exploits. Proper event emission and state updates should also be considered to enhance interoperability and ensure seamless functionality across different DeFi platforms.

RPD - Rebase Price Distortion

Criticality	Minor / Informative
Location	contracts/BUIDL.sol#L360
Status	Unresolved

Description

The rebase mechanism increases the total supply and proportionally adjusts users' balances. Following the rebase, the contract burns the difference in token reserves within the liquidity pair, effectively counteracting the price distortion caused by the rebase's aftereffect. This process restores the token's price to its expected level. However, burning the token's reserve within the pair weakens the reserve supply, as all users' balances have increased while the paired asset's reserves have decreased. This imbalance can lead to heightened price volatility, where subsequent sales may exert a disproportionate impact on the token's price, increasing slippage and potentially discouraging trading activity. Additionally, this practice may introduce vulnerabilities, such as price manipulation or arbitrage opportunities, that traders could exploit.

```
function rebase() private {
    uint256 deltaTime = block.timestamp.sub(lastRebase);
    uint256 times = deltaTime.div(REBASE_INTERVAL);
    uint256 maxTimes = times > 12 ? 12 : times;

    for (uint256 i = 0; i < maxTimes; i++) {
        uint256 supplyDelta =
supply.mul(REBASE_RATE).div(REBASE_RATE_DIVIDER);
        if (supply.add(supplyDelta) >= MAX_SUPPLY)
        {
            supply = MAX_SUPPLY;
            break;
        }
        else
        {
            supply = supply.add(supplyDelta);
        }
        lastRebase = lastRebase.add(REBASE_INTERVAL);
    }
    gonsPerFragment = TOTAL_GONS.div(supply);
    emit LogRebase(block.timestamp, supply);
    manualSkim();
}
```

Recommendation

To mitigate the risks associated with the rebase mechanism, the team could ensure that the liquidity pair is properly provisioned with adequate reserves after each rebase event. By maintaining a balanced reserve ratio between the token and its paired asset, the contract can minimize the potential for price manipulation or arbitrage opportunities that traders could exploit. Proper liquidity provisioning will also help reduce excessive price volatility and slippage, fostering a more stable trading environment. Implementing mechanisms to automatically or manually adjust liquidity after a rebase could further enhance market stability and protect the token's integrity.

PMLPR - Potential Main Liquidity Pair Removal

Criticality	Minor / Informative
Location	contracts/BUIDL.sol#L199
Status	Unresolved

Description

The contract implements transaction restrictions based on the `isLiquidityPair` mapping, which determines whether an address is considered a liquidity pair. If the main liquidity pair is removed from this mapping, interactions with decentralized applications (dApps) such as launchpads, presales, lockers, or staking platforms may be disrupted. These operational restrictions could prevent external contracts from interacting seamlessly with the token, potentially blocking services that rely on its liquidity or trading mechanisms. This issue can hinder token launches, presale events, or staking integrations, limiting the token's usability and adoption within the ecosystem.

```
function updateLiquidityPair(address newPair, bool value) external
onlyOwner {
    require(newPair != address(0), "Zero address");
    require(isLiquidityPair[newPair] != value, "Pair is already the value
of 'value'");

    isLiquidityPair[newPair] = value;
    if (value)
    {
        liquidityPairs.push(newPair);
    }
    else
    {
        for (uint256 i = 0; i < liquidityPairs.length; i++) {
            if (liquidityPairs[i] == newPair) {
                liquidityPairs[i] = liquidityPairs[liquidityPairs.length -
1];
                liquidityPairs.pop();
                break;
            }
        }
    }
    emit NewLiquidityPairUpdated(newPair, value);
}
```

Recommendation

To ensure smooth integration with dApps and external platforms, the contract could prevent the removal of the primary liquidity pair from the `isLiquidityPair` mapping unless an alternative pair is designated to maintain seamless trading operations. By maintaining stability in the `isLiquidityPair` mapping, the contract can ensure broader compatibility with the DeFi ecosystem while preserving its security mechanisms.

TSPL - Total Supply Precision Loss

Criticality	Minor / Informative
Location	contracts/BUIDL.sol#L156
Status	Unresolved

Description

In smart contracts, the total supply represents the total number of tokens created, while individual account balances reflect the portion of that supply owned by each holder. The sum of all account balances should always equal the total supply. However, the contract implements dynamic supply adjustments by calculating the user's balance and total supply by dividing the actual balance with a factor. This calculation could cause a discrepancy between the total supply and the sum of all account balances.

This issue occurs because Solidity uses integer division, which can truncate decimal values potentially leading to precision loss. Such rounding errors may cause inconsistencies, where the total supply does not accurately represent the combined balances of all holders. This may result in unexpected token distribution, incorrect balance calculations, or unintended supply inflation/deflation over time.

```
function balanceOf(address who) public view override returns (uint256) {
    return gonBalances[who].div(gonsPerFragment);
}
```

Recommendation

The total supply and the balance variables are separate and independent from each other. The total supply represents the total number of tokens that have been created, while the balance mapping stores the number of tokens that each account owns. The sum of balances should always equal the total supply.

RSML - Redundant SafeMath Library

Criticality	Minor / Informative
Location	contracts/BUIDL.sol
Status	Unresolved

Description

SafeMath is a popular Solidity library that provides a set of functions for performing common arithmetic operations in a way that is resistant to integer overflows and underflows.

Starting with Solidity versions that are greater than or equal to 0.8.0, the arithmetic operations revert to underflow and overflow. As a result, the native functionality of the Solidity operations replaces the SafeMath library. Hence, the usage of the SafeMath library adds complexity, overhead and increases gas consumption unnecessarily in cases where the explanatory error message is not used.

```
library SafeMath {...}
```

Recommendation

The team is advised to remove the SafeMath library in cases where the revert error message is not used. Since the version of the contract is greater than `0.8.0` then the pure Solidity arithmetic operations produce the same result.

If the previous functionality is required, then the contract could exploit the `unchecked { ... }` statement.

Read more about the breaking change on https://docs.soliditylang.org/en/stable/080-breaking-changes.html#solidity-v0-8-0-breaking-changes.

IDI - Immutable Declaration Improvement

Criticality	Minor / Informative
Location	contracts/BUIDL.sol#L129
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.

```
BNBPair
```

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.

L02 - State Variables could be Declared Constant

Criticality	Minor / Informative
Location	contracts/BUIDL.sol#L85,93,94
Status	Unresolved

Description

State variables can be declared as constant using the constant keyword. This means that the value of the state variable cannot be changed after it has been set. Additionally, the constant variables decrease gas consumption of the corresponding transaction.

```
uint256 public liquiditySellLimit = 1
address public BURN = address(0x000000000000000000000000000000000000dEaD)
address public USDT = address(0x55d398326f99059fF775485246999027B3197955)
```

Recommendation

Constant state variables can be useful when the contract wants to ensure that the value of a state variable cannot be changed by any function in the contract. This can be useful for storing values that are important to the contract's behavior, such as the contract's address or the maximum number of times a certain function can be called. The team is advised to add the constant keyword to state variables that never change.

L04 - Conformance to Solidity Naming Conventions

Criticality	Minor / Informative
Location	contracts/BUIDL.sol#L41,57,93,94,426,453,466
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.

```
function WETH() external pure returns (address);
address private BNBPair
address public BURN = address(0x000000000000000000000000000000000000dEaD)
address public USDT = address(0x55d398326f99059fF775485246999027B3197955)
uint256 BNBAmount
```

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.

L11 - Unnecessary Boolean equality

Criticality	Minor / Informative
Location	contracts/BUIDL.sol#L98
Status	Unresolved

Description

Boolean equality is unnecessary when comparing two boolean values. This is because a boolean value is either true or false, and there is no need to compare two values that are already known to be either true or false.

it's important to be aware of the types of variables and expressions that are being used in the contract's code, as this can affect the contract's behavior and performance. The comparison to boolean constants is redundant. Boolean constants can be used directly and do not need to be compared to true or false.

```
require (inSwap == false, "ReentrancyGuard: reentrant call")
```

Recommendation

Using the boolean value itself is clearer and more concise, and it is generally considered good practice to avoid unnecessary boolean equalities in Solidity code.

L13 - Divide before Multiply Operation

Criticality	Minor / Informative
Location	contracts/BUIDL.sol#L385,386,387,391,397,398,399,404,406,411,416,421
Status	Unresolved

Description

It is important to be aware of the order of operations when performing arithmetic calculations. This is especially important when working with large numbers, as the order of operations can affect the final result of the calculation. Performing divisions before multiplications may cause loss of prediction.

```
uint256 tokenToLiqudity = liquidityFeeTotal.div(2).div(gonsPerFragment)
uint256 swapThreshold = gonSwapThreshold.div(gonsPerFragment)
uint256 liqudityToken =

swapThreshold.mul(tokenToLiqudity).div(tokenToSwap)
```

Recommendation

To avoid this issue, it is recommended to carefully consider the order of operations when performing arithmetic calculations in Solidity. It's generally a good idea to use parentheses to specify the order of operations. The basic rule is that the multiplications should be prior to the divisions.

L16 - Validate Variable Setters

Criticality	Minor / Informative
Location	contracts/BUIDL.sol#L481
Status	Unresolved

Description

The contract performs operations on variables that have been configured on user-supplied input. These variables are missing of proper check for the case where a value is zero. This can lead to problems when the contract is executed, as certain actions may not be properly handled when the value is zero.

```
payable(receiver).transfer(balance)
```

Recommendation

By adding the proper check, the contract will not allow the variables to be configured with zero value. This will ensure that the contract can handle all possible input values and avoid unexpected behavior or errors. Hence, it can help to prevent the contract from being exploited or operating unexpectedly.

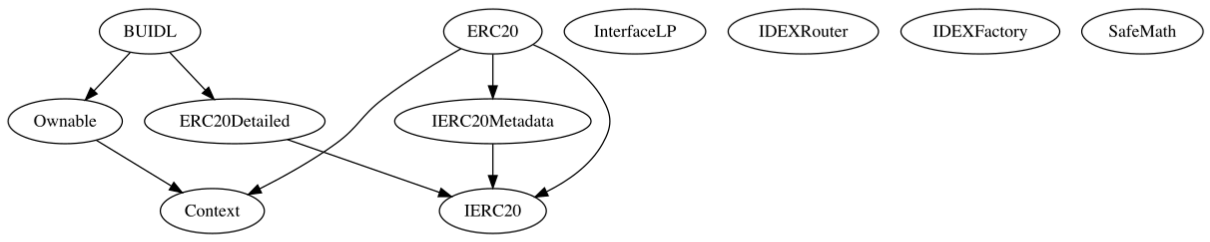
Functions Analysis

Contract	Type	Bases		
	Function Name	Visibility	Mutability	Modifiers
ERC20Detailed	Implementation	IERC20		
		Public	✓	-
	name	Public		-
	symbol	Public		-
	decimals	Public		-
InterfaceLP	Interface			
	token0	External		-
	token1	External		-
	getReserves	External		-
IDEXRouter	Interface			
	factory	External		-
	WETH	External		-
	addLiquidityETH	External	Payable	-
	swapExactTokensForETHSupportingFeeOnTransferTokens	External	✓	-
	swapExactETHForTokensSupportingFeeOnTransferTokens	External	Payable	-
IDEXFactory	Interface			

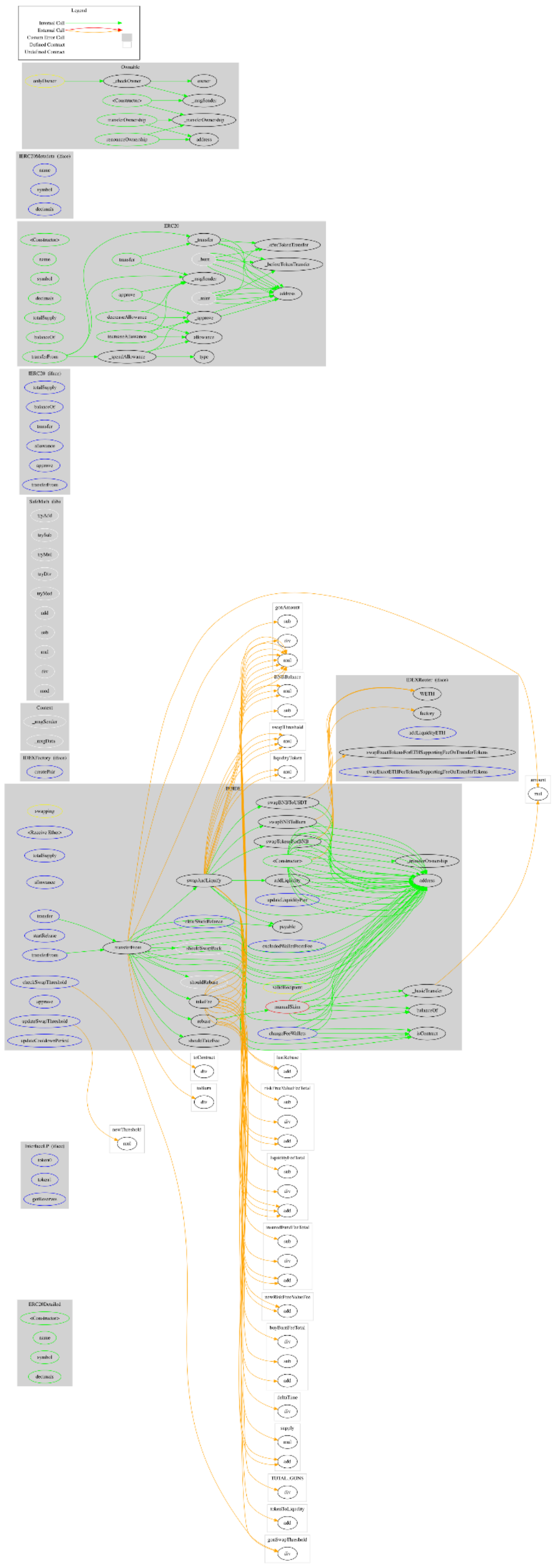
	createPair	External	✓	-
BUIDL	Implementation	ERC20Detailed, Ownable		
		Public	✓	ERC20Detailed
		External	Payable	-
	totalSupply	External		-
	allowance	External		-
	balanceOf	Public		-
	checkSwapThreshold	External		-
	shouldRebase	Internal		
	shouldSwapBack	Internal		
	shouldTakeFee	Internal		
	startRebase	External	✓	onlyOwner
	excludedWalletFromFee	External	✓	onlyOwner
	updateLiquidityPair	External	✓	onlyOwner
	manualSkim	Private	✓	
	transfer	External	✓	validRecipient
	transferFrom	External	✓	validRecipient
	approve	External	✓	-
	_basicTransfer	Internal	✓	
	_transferFrom	Internal	✓	
	takeFee	Internal	✓	
	rebase	Private	✓	

	swapAndLiquify	Private	✓	swapping
	addLiquidity	Private	✓	
	swapTokensForBNB	Private	✓	
	swapBNBToBurn	Private	✓	
	swapBNBToUSDT	Private	✓	
	clearStuckBalance	External	✓	onlyOwner
	changeFeeWallets	External	✓	validRecipient validRecipient validRecipient onlyOwner
	updateSwapThreshold	External	✓	onlyOwner
	updateCooldownPeriod	External	✓	onlyOwner
	isContract	Internal		

Inheritance Graph



Flow Graph



Summary

buidl contract implements a token mechanism. This audit investigates security issues, business logic concerns, and potential improvements. There are some functions that can be abused by the owner like stopping transactions and manipulating the fees. A multi-wallet signing pattern will provide security against potential hacks. Temporarily locking the contract or renouncing ownership will eliminate all the contract threats. The fees are locked at 13% on buy, 15% on sell, and 95%on transfer transactions.

Disclaimer

The information provided in this report does not constitute investment, financial or trading advice and you should not treat any of the document's content as such. This report may not be transmitted, disclosed, referred to or relied upon by any person for any purposes nor may copies be delivered to any other person other than the Company without Cyberscope’s prior written consent. This report is not nor should be considered an “endorsement” or “disapproval” of any particular project or team. This report is not nor should be regarded as an indication of the economics or value of any “product” or “asset” created by any team or project that contracts Cyberscope to perform a security assessment. This document does not provide any warranty or guarantee regarding the absolute bug-free nature of the technology analyzed, nor do they provide any indication of the technologies proprietors' business, business model or legal compliance. This report should not be used in any way to make decisions around investment or involvement with any particular project. This report represents an extensive assessment process intending to help our customers increase the quality of their code while reducing the high level of risk presented by cryptographic tokens and blockchain technology.

Blockchain technology and cryptographic assets present a high level of ongoing risk Cyberscope’s position is that each company and individual are responsible for their own due diligence and continuous security Cyberscope’s goal is to help reduce the attack vectors and the high level of variance associated with utilizing new and consistently changing technologies and in no way claims any guarantee of security or functionality of the technology we agree to analyze. The assessment services provided by Cyberscope are subject to dependencies and are under continuing development. You agree that your access and/or use including but not limited to any services reports and materials will be at your sole risk on an as-is where-is and as-available basis Cryptographic tokens are emergent technologies and carry with them high levels of technical risk and uncertainty. The assessment reports could include false positives false negatives and other unpredictable results. The services may access and depend upon multiple layers of third parties.

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