



Cyberscope

Audit Report

Potter Pigl

February 2023

Type BEP20

Network BSC

Address 0x3fd7ee4a8ae0389763b080c871e1827b167c3cbd

Audited by © cyberscope

Table of Contents

Table of Contents	1
Review	3
Audit Updates	3
Source Files	3
Analysis	4
ELFM - Exceeds Fees Limit	5
Description	5
Recommendation	5
Diagnostics	6
ZD - Zero Division	7
Description	7
Recommendation	7
PVC - Price Volatility Concern	8
Description	8
Recommendation	8
PTRP - Potential Transfer Revert Propagation	9
Description	9
Recommendation	9
L02 - State Variables could be Declared Constant	10
Description	10
Recommendation	10
L04 - Conformance to Solidity Naming Conventions	11
Description	11
Recommendation	12
L07 - Missing Events Arithmetic	13
Description	13
Recommendation	13
L09 - Dead Code Elimination	14
Description	14
Recommendation	14
L16 - Validate Variable Setters	16
Description	16

Recommendation	16
L17 - Usage of Solidity Assembly	17
Description	17
Recommendation	17
L19 - Stable Compiler Version	18
Description	18
Recommendation	18
Functions Analysis	19
Inheritance Graph	24
Flow Graph	25
Summary	26
Disclaimer	27
About Cyberscope	28

Review

Contract Name	PPIG
Compiler Version	v0.8.17+commit.8df45f5f
Optimization	200 runs
Explorer	https://bscscan.com/address/0x3fd7ee4a8ae0389763b080c871e1827b167c3cbd
Address	0x3fd7ee4a8ae0389763b080c871e1827b167c3cbd
Network	BSC
Symbol	PPIG
Decimals	9
Total Supply	1,000,000

Audit Updates

Initial Audit	12 Feb 2023
---------------	-------------

Source Files

Filename	SHA256
PPIG.sol	e3b8cee77655687b2755fccf69c44fe418cbd0a9402faba56cd8910b9688b6b4

Analysis

● Critical ● Medium ● Minor / Informative ● Pass

Severity	Code	Description	Status
●	ST	Stops Transactions	Passed
●	OCTD	Transfers Contract's Tokens	Passed
●	OTUT	Transfers User's Tokens	Passed
●	ELFM	Exceeds Fees Limit	Unresolved
●	ULTW	Transfers Liquidity to Team Wallet	Passed
●	MT	Mints Tokens	Passed
●	BT	Burns Tokens	Passed
●	BC	Blacklists Addresses	Passed

ELFM - Exceeds Fees Limit

Criticality	Critical
Location	PPIG.sol#L526,533
Status	Unresolved

Description

The contract owner has the authority to increase over the allowed limit of 25%. The owner may take advantage of it by calling the `setSellTaxes` function with a high percentage value. As a result, the fees may be configured up to 50%.

```
function setSellTaxes(uint256 sellMarketingFeeNew, uint256 sellBurnFeeNew)
external onlyOwner(){
    sellMarketingFee = sellMarketingFeeNew;
    sellBurnFee = sellBurnFeeNew;
    totalTaxIfSelling = sellMarketingFee.add(sellBurnFee);
    require(totalTaxIfSelling <= 50 , "Max sell fee under 50%");
}
```

Recommendation

The contract could embody a check for the maximum acceptable value. 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. That risk can be prevented by temporarily locking the contract or renouncing ownership.

Diagnostics

● Critical ● Medium ● Minor / Informative

Severity	Code	Description	Status
●	ZD	Zero Division	Unresolved
●	PVC	Price Volatility Concern	Unresolved
●	PTRP	Potential Transfer Revert Propagation	Unresolved
●	L02	State Variables could be Declared Constant	Unresolved
●	L04	Conformance to Solidity Naming Conventions	Unresolved
●	L07	Missing Events Arithmetic	Unresolved
●	L09	Dead Code Elimination	Unresolved
●	L16	Validate Variable Setters	Unresolved
●	L17	Usage of Solidity Assembly	Unresolved
●	L19	Stable Compiler Version	Unresolved

ZD - Zero Division

Criticality	Medium
Location	PPIG.sol#L540
Status	Unresolved

Description

The contract is using variables that may be set to zero as denominators. This can lead to unpredictable and potentially harmful results, such as a transaction revert.

```
uint256 burnAmount = tAmount.mul(burnShare).div(totalDistributionShares);
```

Recommendation

It is important to handle division by zero appropriately in the code to avoid unintended behavior and to ensure the reliability and safety of the contract. The contract should ensure that the divisor is always non-zero before performing a division operation. It should prevent the variables to be set to zero or should not allow executing of the corresponding statements.

PVC - Price Volatility Concern

Criticality	Minor / Informative
Location	PPIG.sol#L522
Status	Unresolved

Description

The contract accumulates tokens from the taxes to swap them for ETH. The variable `minimumTokensBeforeSwap` 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 setNumTokensBeforeSwap(uint256 newLimit) external onlyOwner() {  
    minimumTokensBeforeSwap = newLimit;  
}
```

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 total supply. Hence, the contract will guarantee that it cannot accumulate a huge amount of tokens in order to sell them.

PTRP - Potential Transfer Revert Propagation

Criticality	Minor / Informative
Location	PPIG.sol#L545
Status	Unresolved

Description

The contract sends funds to a `marketingWallet` 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.

```
function setMktWallet(address payable account) external onlyOwner{
    mktWallet = account;
}
```

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 archived by not allowing set contract addresses or by sending the funds in a non-revertable way.

L02 - State Variables could be Declared Constant

Criticality	Minor / Informative
Location	PPIG.sol#L376,377,378,382,412,413
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.

```
string private _name = "Potter Pig"
string private _symbol = "PPIG"
uint8 private _decimals = 9
address public deadAddress = 0x00000000000000000000000000000000dEaD
bool public swapAndLiquifyEnabled = true
bool public swapAndLiquifyByLimitOnly = false
```

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	PPIG.sol#L202,203,219,238,384,536,545
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 DOMAIN_SEPARATOR() external view returns (bytes32);
function PERMIT_TYPEHASH() external pure returns (bytes32);
function MINIMUM_LIQUIDITY() external pure returns (uint);
function WETH() external pure returns (address);
mapping (address => uint256) _balances
uint256 _marketingShareNew

function GoTrade() external onlyOwner{
    trade = true;
}
```

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/v0.8.17/style-guide.html#naming-convention>.

L07 - Missing Events Arithmetic

Criticality	Minor / Informative
Location	PPIG.sol#L519,523,530,538
Status	Unresolved

Description

Events are a way to record and log information about changes or actions that occur within a contract. They are often used to notify external parties or clients about events that have occurred within the contract, such as the transfer of tokens or the completion of a task.

It's important to carefully design and implement the events in a contract, and to ensure that all required events are included. It's also a good idea to test the contract to ensure that all events are being properly triggered and logged.

```
minimumTokensBeforeSwap = newLimit  
buyMarketingFee = buyMarketingFeeNew  
sellMarketingFee = sellMarketingFeeNew  
burnShare = burnShareNew
```

Recommendation

By including all required events in the contract and thoroughly testing the contract's functionality, the contract ensures that it performs as intended and does not have any missing events that could cause issues with its arithmetic.

L09 - Dead Code Elimination

Criticality	Minor / Informative
Location	PPIG.sol#L83,94,102,106,110,114,119
Status	Unresolved

Description

In Solidity, dead code is code that is written in the contract, but is never executed or reached during normal contract execution. Dead code can occur for a variety of reasons, such as:

- Conditional statements that are always false.
- Functions that are never called.
- Unreachable code (e.g., code that follows a return statement).

Dead code can make a contract more difficult to understand and maintain, and can also increase the size of the contract and the cost of deploying and interacting with it.

```
function isContract(address account) internal view returns (bool) {
    // According to EIP-1052, 0x0 is the value returned for not-yet
    created accounts
    // and
    0xc5d2460186f7233c927e7db2dcc703c0e500b653ca82273b7bfad8045d85a470 is returned
    // for accounts without code, i.e. `keccak256('')`
    bytes32 codehash;
    bytes32 accountHash =
    0xc5d2460186f7233c927e7db2dcc703c0e500b653ca82273b7bfad8045d85a470;
    // solhint-disable-next-line no-inline-assembly
    assembly { codehash := extcodehash(account) }
    return (codehash != accountHash && codehash != 0x0);
}

...
```

Recommendation

To avoid creating dead code, it's important to carefully consider the logic and flow of the contract and to remove any code that is not needed or that is never executed. This can help improve the clarity and efficiency of the contract.

L16 - Validate Variable Setters

Criticality	Minor / Informative
Location	PPIG.sol#L542
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.

```
mktWallet = account
```

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.

L17 - Usage of Solidity Assembly

Criticality	Minor / Informative
Location	PPIG.sol#L90,128
Status	Unresolved

Description

Using assembly can be useful for optimizing code, but it can also be error-prone. It's important to carefully test and debug assembly code to ensure that it is correct and does not contain any errors.

Some common types of errors that can occur when using assembly in Solidity include Syntax, Type, Out-of-bounds, Stack, and Revert.

```
assembly { codehash := extcodehash(account) }

assembly {
    let returndata_size := mload(returndata)
    revert(add(32, returndata), returndata_size)
}
```

Recommendation

It is recommended to use assembly sparingly and only when necessary, as it can be difficult to read and understand compared to Solidity code.

L19 - Stable Compiler Version

Criticality	Minor / Informative
Location	PPIG.sol#L2
Status	Unresolved

Description

The `^` symbol indicates that any version of Solidity that is compatible with the specified version (i.e., any version that is a higher minor or patch version) can be used to compile the contract. The version lock is a mechanism that allows the author to specify a minimum version of the Solidity compiler that must be used to compile the contract code. This is useful because it ensures that the contract will be compiled using a version of the compiler that is known to be compatible with the code.

```
pragma solidity ^0.8.0;
```

Recommendation

The team is advised to lock the pragma to ensure the stability of the codebase. The locked pragma version ensures that the contract will not be deployed with an unexpected version. An unexpected version may produce vulnerabilities and undiscovered bugs. The compiler should be configured to the lowest version that provides all the required functionality for the codebase. As a result, the project will be compiled in a well-tested LTS (Long Term Support) environment.

Functions Analysis

Contract	Type	Bases		
	Function Name	Visibility	Mutability	Modifiers
Context	Implementation			
	_msgSender	Internal		
	_msgData	Internal		
IERC20	Interface			
	totalSupply	External		-
	balanceOf	External		-
	transfer	External	✓	-
	allowance	External		-
	approve	External	✓	-
	transferFrom	External	✓	-
SafeMath	Library			
	add	Internal		
	sub	Internal		
	sub	Internal		
	mul	Internal		
	div	Internal		
	div	Internal		
	mod	Internal		
	mod	Internal		
Address	Library			
	isContract	Internal		

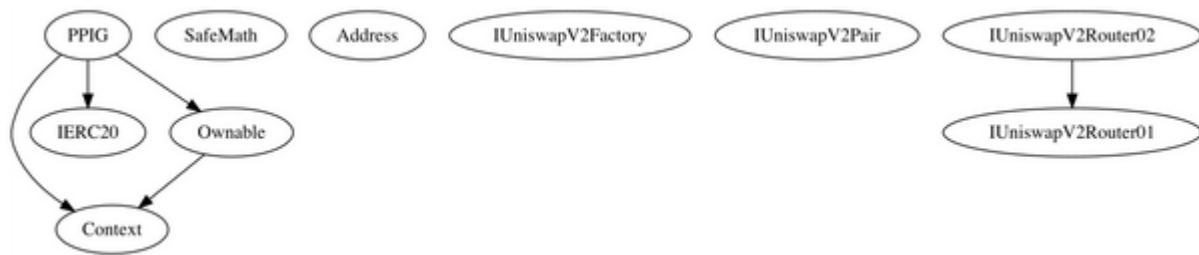
	sendValue	Internal	✓	
	functionCall	Internal	✓	
	functionCall	Internal	✓	
	functionCallWithValue	Internal	✓	
	functionCallWithValue	Internal	✓	
	_functionCallWithValue	Private	✓	
Ownable	Implementation	Context		
		Public	✓	-
	owner	Public		-
	waiveOwnership	Public	✓	onlyOwner
	transferOwnership	Public	✓	onlyOwner
IUniswapV2Factory	Interface			
	feeTo	External		-
	feeToSetter	External		-
	getPair	External		-
	allPairs	External		-
	allPairsLength	External		-
	createPair	External	✓	-
	setFeeTo	External	✓	-
	setFeeToSetter	External	✓	-
IUniswapV2Pair	Interface			
	name	External		-
	symbol	External		-
	decimals	External		-
	totalSupply	External		-
	balanceOf	External		-

	allowance	External		-
	approve	External	✓	-
	transfer	External	✓	-
	transferFrom	External	✓	-
	DOMAIN_SEPARATOR	External		-
	PERMIT_TYPEHASH	External		-
	nonces	External		-
	permit	External	✓	-
	MINIMUM_LIQUIDITY	External		-
	factory	External		-
	token0	External		-
	token1	External		-
	getReserves	External		-
	price0CumulativeLast	External		-
	price1CumulativeLast	External		-
	kLast	External		-
	burn	External	✓	-
	swap	External	✓	-
	skim	External	✓	-
	sync	External	✓	-
	initialize	External	✓	-
IUniswapV2Router01	Interface			
	factory	External		-
	WETH	External		-
	addLiquidity	External	✓	-
	addLiquidityETH	External	Payable	-
	removeLiquidity	External	✓	-
	removeLiquidityETH	External	✓	-

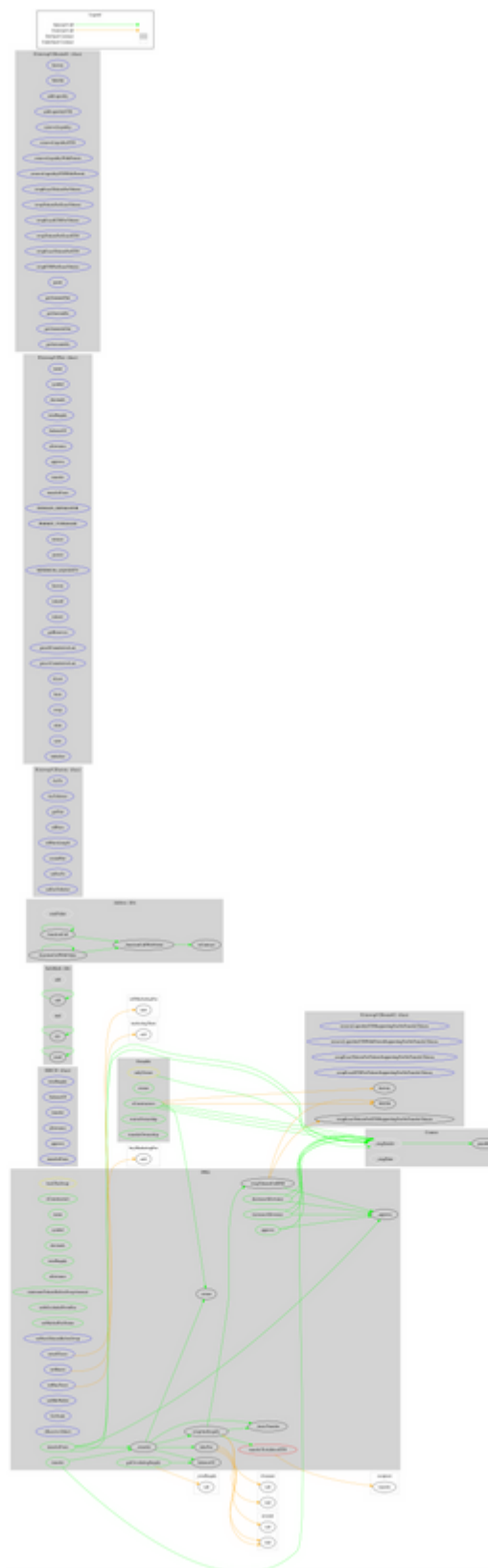
	removeLiquidityWithPermit	External	✓	-
	removeLiquidityETHWithPermit	External	✓	-
	swapExactTokensForTokens	External	✓	-
	swapTokensForExactTokens	External	✓	-
	swapExactETHForTokens	External	Payable	-
	swapTokensForExactETH	External	✓	-
	swapExactTokensForETH	External	✓	-
	swapETHForExactTokens	External	Payable	-
	quote	External		-
	getAmountOut	External		-
	getAmountIn	External		-
	getAmountsOut	External		-
	getAmountsIn	External		-
IUniswapV2Router02	Interface	IUniswapV2Router01		
	removeLiquidityETHSupportingFeeOnTransferTokens	External	✓	-
	removeLiquidityETHWithPermitSupportingFeeOnTransferTokens	External	✓	-
	swapExactTokensForTokensSupportingFeeOnTransferTokens	External	✓	-
	swapExactETHForTokensSupportingFeeOnTransferTokens	External	Payable	-
	swapExactTokensForETHSupportingFeeOnTransferTokens	External	✓	-
PPIG	Implementation	Context, IERC20, Ownable		
		Public	✓	-
	name	Public		-
	symbol	Public		-
	decimals	Public		-

	totalSupply	Public		-
	balanceOf	Public		-
	allowance	Public		-
	increaseAllowance	Public	✓	-
	decreaseAllowance	Public	✓	-
	minimumTokensBeforeSwapAmount	Public		-
	approve	Public	✓	-
	_approve	Private	✓	
	setIsExcludedFromFee	Public	✓	onlyOwner
	setMarketPairStatus	Public	✓	onlyOwner
	setNumTokensBeforeSwap	External	✓	onlyOwner
	setBuyTaxes	External	✓	onlyOwner
	setSellTaxes	External	✓	onlyOwner
	setShares	External	✓	onlyOwner
	setMktWallet	External	✓	onlyOwner
	GoTrade	External	✓	onlyOwner
	getCirculatingSupply	Public		-
	transferToAddressETH	Private	✓	
		External	Payable	-
	transfer	Public	✓	-
	transferFrom	Public	✓	-
	_transfer	Private	✓	
	_basicTransfer	Internal	✓	
	swapAndLiquify	Private	✓	lockTheSwap
	swapTokensForBNB	Private	✓	
	takeFee	Internal	✓	

Inheritance Graph



Flow Graph



Summary

There are some functions that can be abused by the owner like manipulate 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.

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

<https://www.cyberscope.io>