Cyberscope

Audit Report

PussFi

December 2024

Network	TRON
Address	TX5eXdf8458bZ77fk8xdvUgiQmC3L93iv7
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
●	GO	Gas Optimization	Unresolved
●	MEE	Missing Events Emission	Unresolved
●	MSC	Missing Sanity Check	Unresolved
●	RID	Redundant Interface Declaration	Unresolved
●	VTO	Variable Type Optimization	Unresolved
●	L04	Conformance to Solidity Naming Conventions	Unresolved
●	L07	Missing Events Arithmetic	Unresolved
●	L09	Dead Code Elimination	Unresolved
●	L15	Local Scope Variable Shadowing	Unresolved
●	L18	Multiple Pragma Directives	Unresolved
●	L19	Stable Compiler Version	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	Token
Compiler Version	v0.8.6+commit.11564f7e
Optimization	200 runs
Explorer	https://tronscan.org/#/token20/TX5eXdf8458bZ77fk8xdvUgiQmC3L93iv7
Address	TX5eXdf8458bZ77fk8xdvUgiQmC3L93iv7
Network	TRON
Symbol	PUSS
Decimals	18
Total Supply	1,000,000,000
Badge Eligibility	Yes

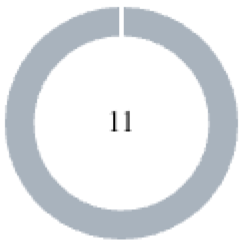
Audit Updates

Initial Audit	17 Dec 2024

Source Files

Filename	SHA256
token.sol	586f45c92ddac617874bbb0a3ba04ecf030b863c8421291664c95f7591a1f73c

Findings Breakdown



●	Critical	0
●	Medium	0
●	Minor / Informative	11

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

GO - Gas Optimization

Criticality	Minor / Informative
Location	token.sol#L668
Status	Unresolved

Description

Gas optimization refers to the process of reducing the amount of gas required to execute a transaction. Gas is the unit of measurement used to calculate the fees paid to miners for including a transaction in a block on the blockchain.

The contract modifies the state of certain variables when the provided argument is different than a predefined state. However, in the case where the argument matches the predefined state of the variable, the contract will not modify the state but the caller of the function will still be charged with gas.

```
function setMode(uint v) public onlyOwner {
    if (_mode != MODE_NORMAL) {
        _mode = v;
    }
}
```

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.

The contract could modify these segments to use the `require` function provided by Solidity. By doing so, if the condition is not met, the transaction will revert and the caller will not be charged for executing the function.

MEE - Missing Events Emission

Criticality	Minor / Informative
Location	token.sol#L670
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.

```
_mode = v;
```

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.

MSC - Missing Sanity Check

Criticality	Minor / Informative
Location	token.sol#L668
Status	Unresolved

Description

The contract is processing variables that have not been properly sanitized and checked that they form the proper shape. These variables may produce vulnerability issues.

For instance, the `_mode` variable is intended to accept three possible values (0,1,2). However, the function does not ensure this restriction.

```
function setMode(uint v) public onlyOwner {
    if (_mode != MODE_NORMAL) {
        _mode = v;
    }
}
```

Recommendation

The team is advised to properly check the variables according to the required specifications.

RID - Redundant Interface Declaration

Criticality	Minor / Informative
Location	token.sol#L631
Status	Unresolved

Description

There are code segments that could be optimized. A segment may be optimized so that it becomes a smaller size, consumes less memory, executes more rapidly, or performs fewer operations.

The contract declares certain interfaces that are not used in a meaningful way by the contract. As a result, these interfaces are redundant.

```
interface IProxy {
    function _acceptImplementation() external;
    function admin() external view returns (address);
}
```

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.

VTO - Variable Type Optimization

Criticality	Minor / Informative
Location	token.sol#L620,621,622,623
Status	Unresolved

Description

The contract declares certain variables as `uint256`, even though their maximum value is capped at 2. By performing a mathematical analysis, it becomes evident that the highest value, 2, can be comfortably stored in a `uint8` type, as the logarithm base-2 of 2 results in 1. As a result, the contracts reserve unnecessary storage space and consume more gas when using these variables.

```
uint public constant MODE_NORMAL = 0;
uint public constant MODE_TRANSFER_RESTRICTED = 1;
uint public constant MODE_TRANSFER_CONTROLLED = 2;
uint public _mode;
```

Recommendation

To optimize the smart contract and improve resource utilization, it is strongly recommended to review and update the variable types used. Specifically, consider changing variables currently declared as `uint256` to `uint8` wherever applicable, given that the maximum value they store is 2. This adjustment aligns the variable types more closely with the actual data requirements, reducing unnecessary gas costs and optimizing storage efficiency.

L04 - Conformance to Solidity Naming Conventions

Criticality	Minor / Informative
Location	token.sol#L623,632
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.

```
uint public _mode
function _acceptImplementation() external;
```

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.

L07 - Missing Events Arithmetic

Criticality	Minor / Informative
Location	token.sol#L670
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.

```
_mode = v
```

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	token.sol#L424
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 _burn(address account, uint256 amount) internal virtual {
        require(account != address(0), "ERC20: burn from the zero
address");

        _beforeTokenTransfer(account, address(0), amount);

        uint256 accountBalance = _balances[account];
...
            _totalSupply -= amount;
        }

        emit Transfer(account, address(0), amount);

        _afterTokenTransfer(account, address(0), amount);
    }
```

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.

L15 - Local Scope Variable Shadowing

Criticality	Minor / Informative
Location	token.sol#L647,648,649
Status	Unresolved

Description

Local scope variable shadowing occurs when a local variable with the same name as a variable in an outer scope is declared within a function or code block. When this happens, the local variable "shadows" the outer variable, meaning that it takes precedence over the outer variable within the scope in which it is declared.

```
string memory name
string memory symbol
uint256 totalSupply
```

Recommendation

It's important to be aware of shadowing when working with local variables, as it can lead to confusion and unintended consequences if not used correctly. It's generally a good idea to choose unique names for local variables to avoid shadowing outer variables and causing confusion.

L18 - Multiple Pragma Directives

Criticality	Minor / Informative
Location	token.sol#L4,89,118,145,535,617,629,639
Status	Unresolved

Description

If the contract includes multiple conflicting pragma directives, it may produce unexpected errors. To avoid this, it's important to include the correct pragma directive at the top of the contract and to ensure that it is the only pragma directive included in the contract.

```
pragma solidity ^0.8.0;
pragma solidity 0.8.6;
```

Recommendation

It is important to include only one pragma directive at the top of the contract and to ensure that it accurately reflects the version of Solidity that the contract is written in.

By including all required compiler options and flags in a single pragma directive, the potential conflicts could be avoided and ensure that the contract can be compiled correctly.

L19 - Stable Compiler Version

Criticality	Minor / Informative
Location	token.sol#L89
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.

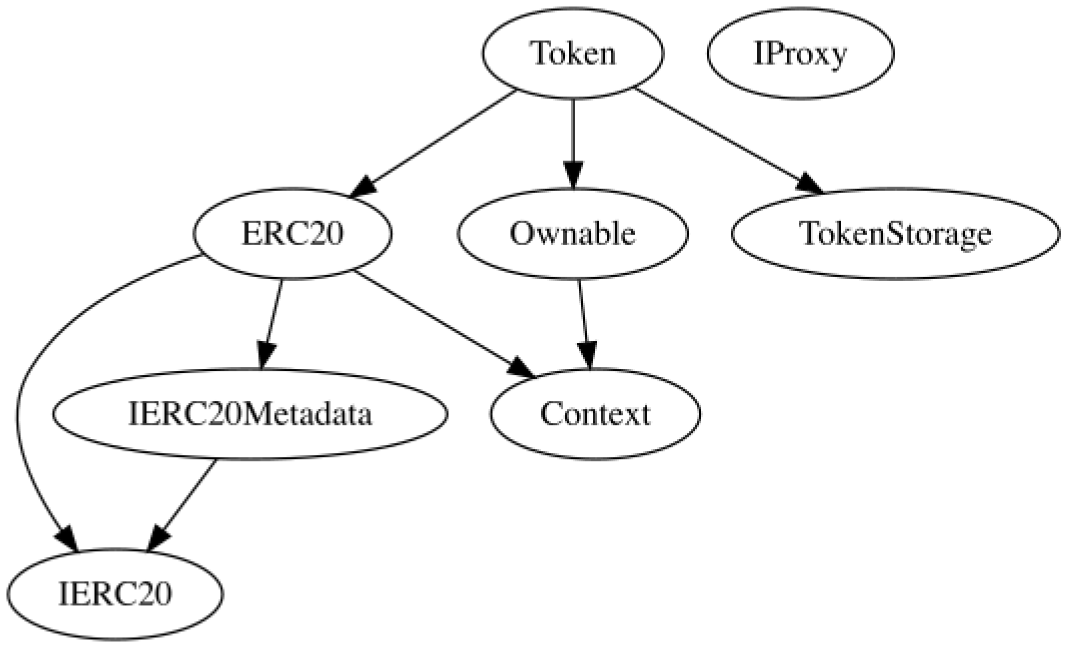
Functions Analysis

Contract	Type	Bases		
	Function Name	**Visibility**	**Mutability**	**Modifiers**
IERC20	Interface			
	totalSupply	External		-
	balanceOf	External		-
	transfer	External	✓	-
	allowance	External		-
	approve	External	✓	-
	transferFrom	External	✓	-
IERC20Metadata	Interface	IERC20		
	name	External		-
	symbol	External		-
	decimals	External		-
Context	Implementation			
	_msgSender	Internal		
	_msgData	Internal		
ERC20	Implementation	Context, IERC20, IERC20Metadata		
		Public	✓	-

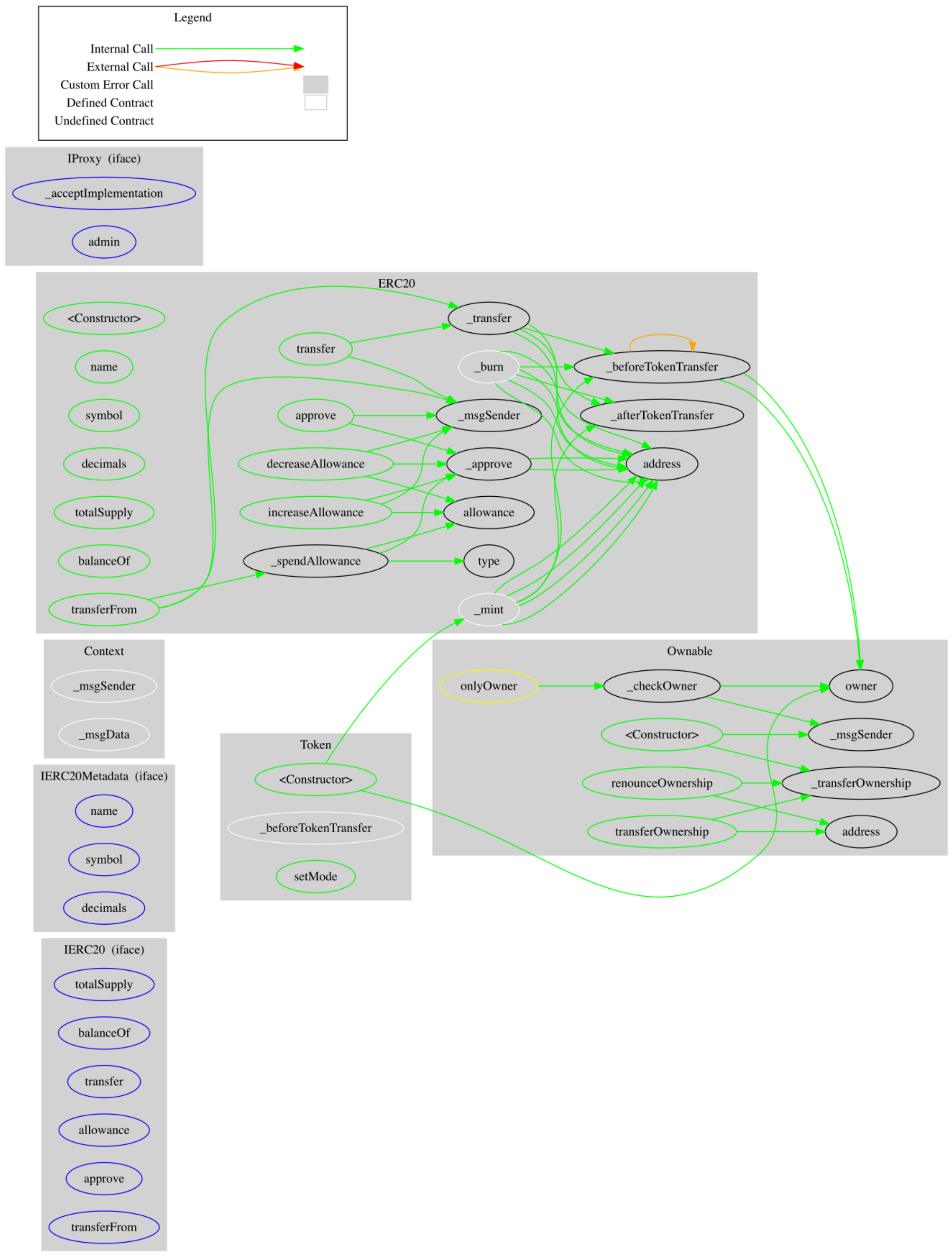
	name	Public		-
	symbol	Public		-
	decimals	Public		-
	totalSupply	Public		-
	balanceOf	Public		-
	transfer	Public	✓	-
	allowance	Public		-
	approve	Public	✓	-
	transferFrom	Public	✓	-
	increaseAllowance	Public	✓	-
	decreaseAllowance	Public	✓	-
	_transfer	Internal	✓	
	_mint	Internal	✓	
	_burn	Internal	✓	
	_approve	Internal	✓	
	_spendAllowance	Internal	✓	
	_beforeTokenTransfer	Internal	✓	
	_afterTokenTransfer	Internal	✓	
Ownable	Implementation	Context		
		Public	✓	-
	owner	Public		-
	_checkOwner	Internal		
	renounceOwnership	Public	✓	onlyOwner
	transferOwnership	Public	✓	onlyOwner

	_transferOwnership	Internal	✓	
TokenStorage	Implementation			
IProxy	Interface			
	_acceptImplementation	External	✓	-
	admin	External		-
Token	Implementation	ERC20, Ownable, TokenStorage		
		Public	✓	ERC20
	_beforeTokenTransfer	Internal	✓	
	setMode	Public	✓	onlyOwner

Inheritance Graph



Flow Graph


Legend
Internal Call
External Call
Custom Error Call
Defined Contract
Undefined Contract
IProxy (iface)
_acceptImplementation
admin
ERC20
<Constructor>
name
symbol
decimals
totalSupply
balanceOf
transferFrom
transfer
approve
decreaseAllowance
increaseAllowance
_spendAllowance
_transfer
_burn
_msgSender
_approve
allowance
type
_mint
_beforeTokenTransfer
_afterTokenTransfer
address
Context
_msgSender
_msgData
IERC20Metadata (iface)
name
symbol
decimals
IERC20 (iface)
totalSupply
balanceOf
transfer
allowance
approve
transferFrom
Token
<Constructor>
_beforeTokenTransfer
setMode
Ownable
onlyOwner
_checkOwner
owner
<Constructor>
_msgSender
renounceOwnership
_transferOwnership
transferOwnership
address

Summary

PussFi contract implements a token mechanism. This audit investigates security issues, business logic concerns, and potential improvements. PussFi is an interesting project that has a friendly and growing community. The Smart Contract analysis reported no compiler errors or critical issues. The contract Owner can access some admin functions that can not be used in a malicious way to disturb the users' transactions.

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