SOLIDCHECK

STATIC AUDITING REPORT FOR:

MyNewToken

Creation-Date: 19/04/2023

Table of Content

Disclaimer 3
Project Overview 4
Source Units In Scope 5
Overview 6
Callgraph 7
Inheritance Graph 8
Issue Count 9
Risk Chart 10
Source Lines (sloc vs nsloc) 11
Risks and Vulnerabilities 12
Found Vulnerabilities 13
Contract Summary 21

Disclaimer

The audit makes no statements or warrantees about utility of the code, safety of the code, suitability of the business model, regulatory regime for the business model, or any other statements about fitness of the contracts to purpose, or their bug free status. The audit documentation is for discussion purposes only.
The information presented in this report is confidential and privileged. If you are reading this report, you agree to keep it confidential, not to copy, disclose or disseminate without the agreement of "MyNewToken".
If you are not the intended receptor of this document, remember that any disclosure, copying or dissemination of it is forbidden. The Solidcheck audit is generated automatically and not created by the provider of the service!

Project Overview

TheMyToken contract is an intentionally insecure implementation of a basic ERC20 token with multiple vulnerabilities, including a reentrancy attack, an unhashed private password variable, missing natspec documentation, authorization through tx.origin, and an outdated Solidity version. This contract serves as an educational example to demonstrate Solidcheck´s potential to find risks in smart contracts and should not be used in real-world applications.

twitter: https://twitter.com/MyNewExampleToken

linkedin: https://linkedin.com/in/ProjectName

facebook: https://facebook.com/TokenName

github: https://github.com/YourGithubURL

youtube: https://youtube.com/YourYOutubeURL

Source Units In Scope

FILE	LOGIC CONTRACTS	INTERFACES	LINES	NLINES	NSLOC	COMMENT LINES	COMPLEX. SCORE
[redacted]/Code/solidcheck/backenMyToken.sol	1	---	122	105	81	7	55
Totals	1	---	122	105	81	7	55

Overview

Exposed Functions

PUBLIC	PAYABLE
8	0

EXTERNAL	INTERNAL	PRIVATE	PURE	VIEW
8	10	0	0	3

State Variables

TOTAL	PUBLIC
8	5

Capabilities

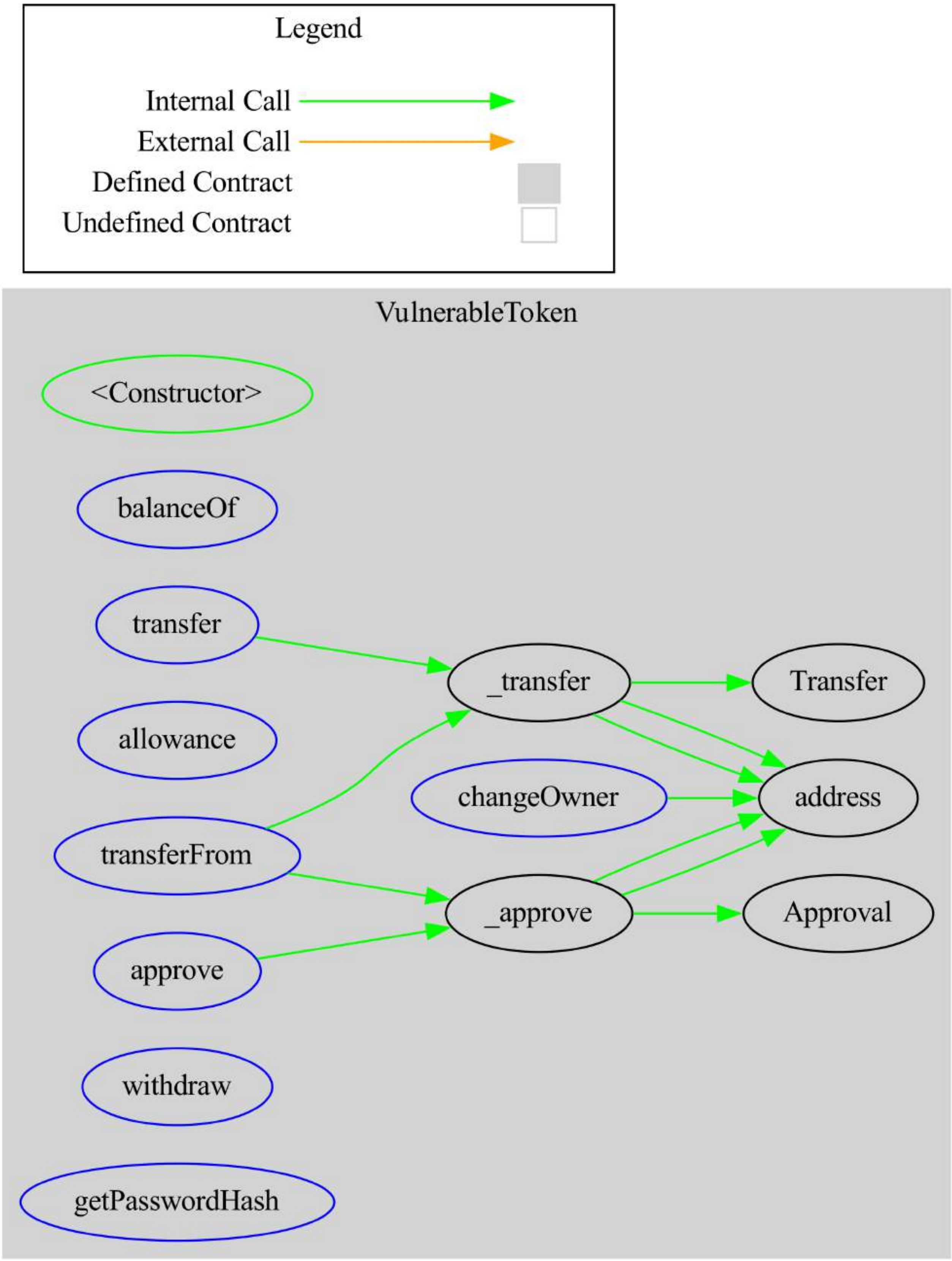
SOLIDITY VERSIONS OBSERVED	EXPERIMENTAL FEATURES	CAN RECEIVE FUNDS	USES ASSEMBLY	HAS DESTROYABLE CONTRACTS
^0.7.0		---	---	---

TRANSFERS ETH	LOW-LEVEL CALLS	DELEGATECALL	USES HASH FUNCTIONS	ECRECOVER	NEW/CREATE/CREATE2
---	---	---	yes	---	---

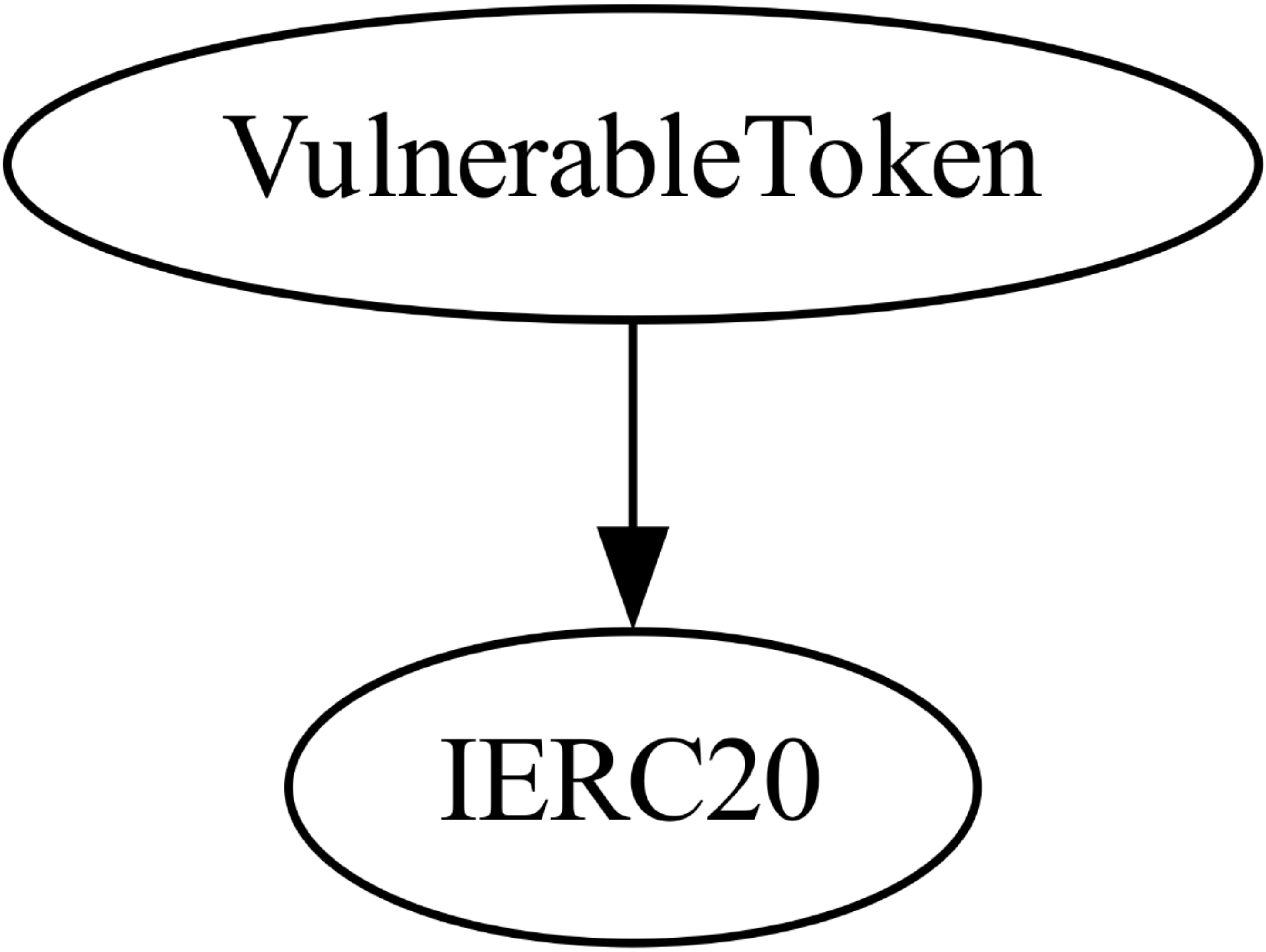
Dependencies

DEPENDENCY / IMPORT PATH	COUNT
https://github.com/OpenZeppelin/openzeppelin-contracts/blob/v3.4.0/contracts/math/SafeMath.sol	1
https://github.com/OpenZeppelin/openzeppelin-contracts/blob/v3.4.0/contracts/token/ERC20/IERC20.sol	1

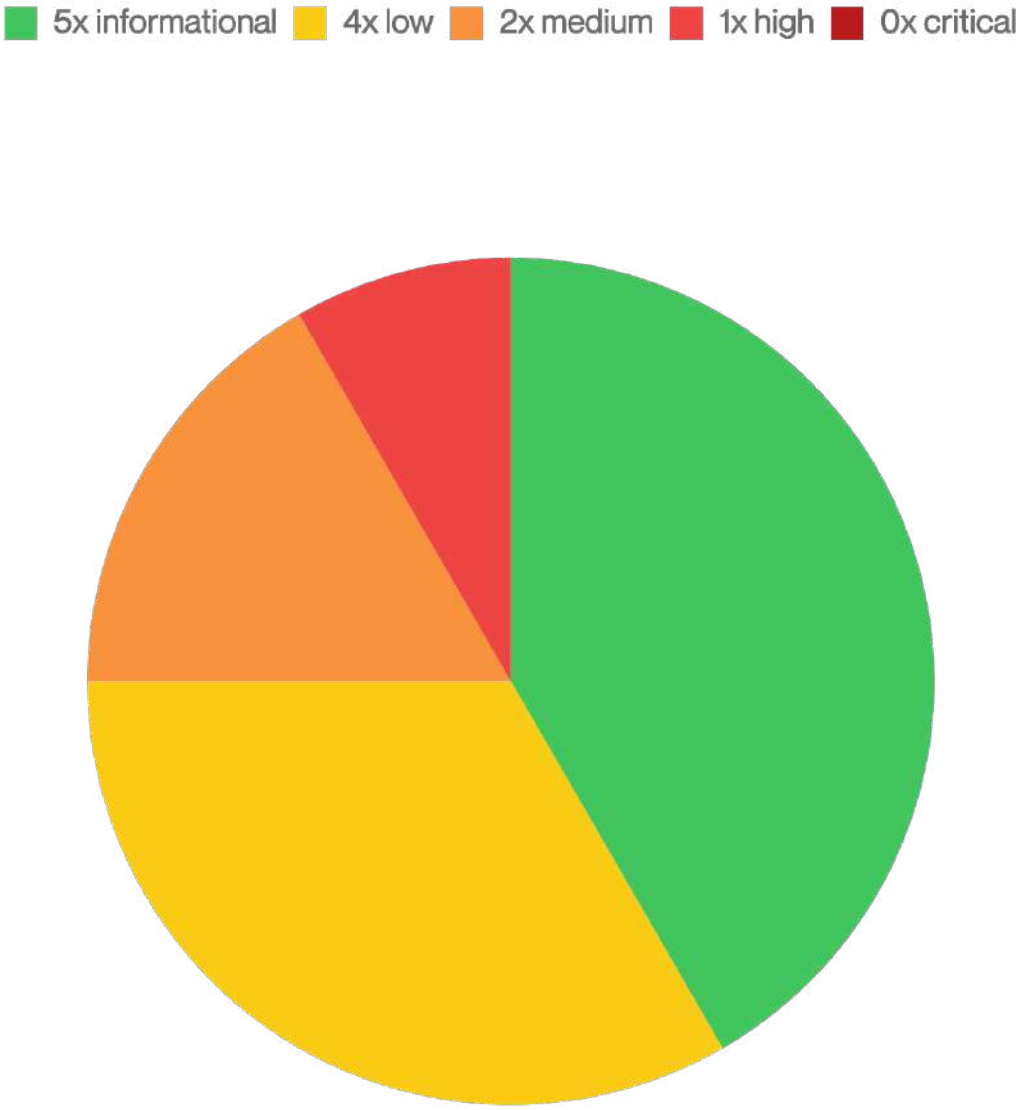
Callgraph


Legend
Internal Call
External Call
Defined Contract
Undefined Contract
VulnerableToken
<Constructor>
balanceOf
transfer
allowance
transferFrom
approve
withdraw
getPasswordHash
_transfer
changeOwner
_approve
Transfer
address
Approval

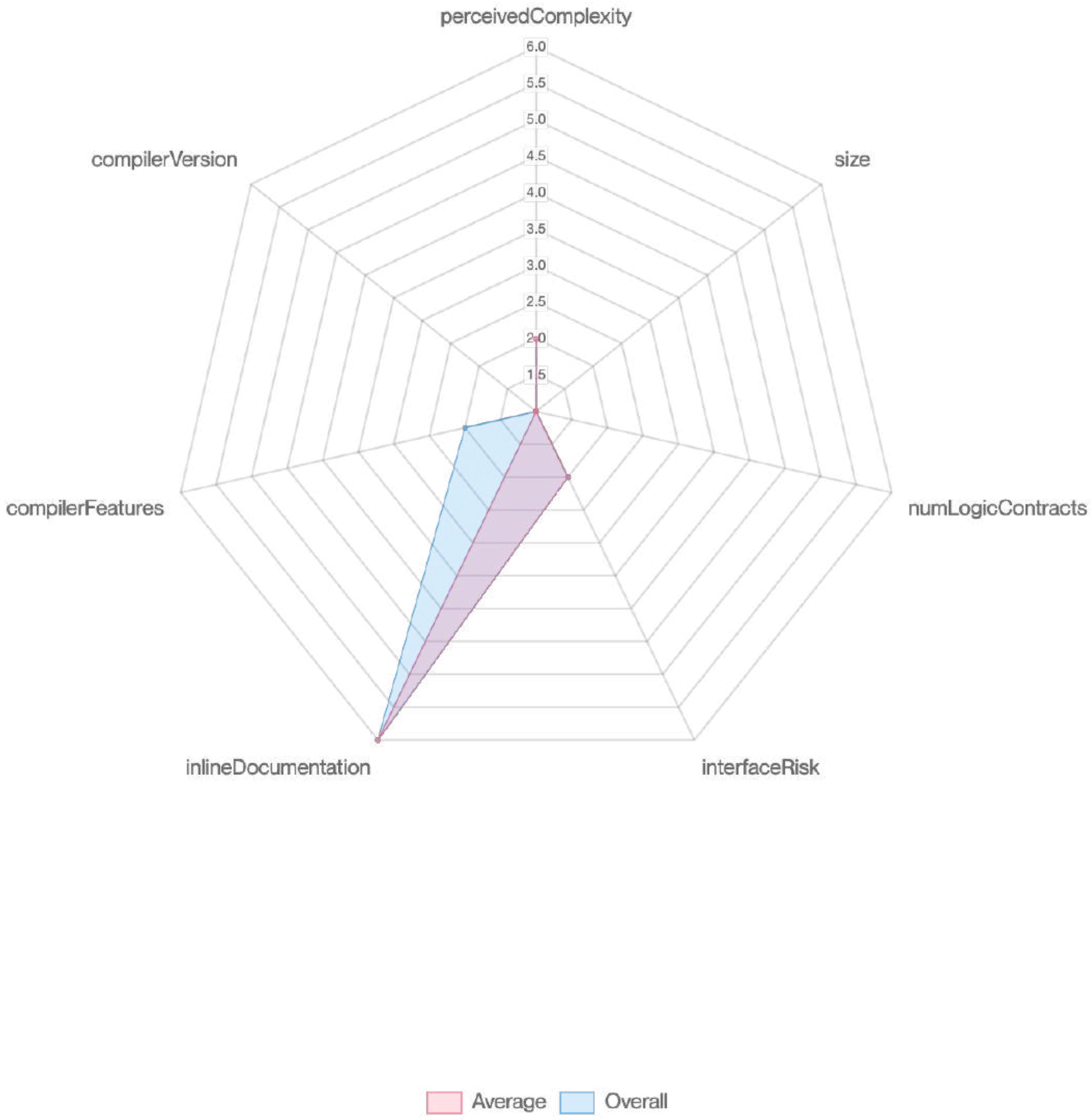
Inheritance Graph



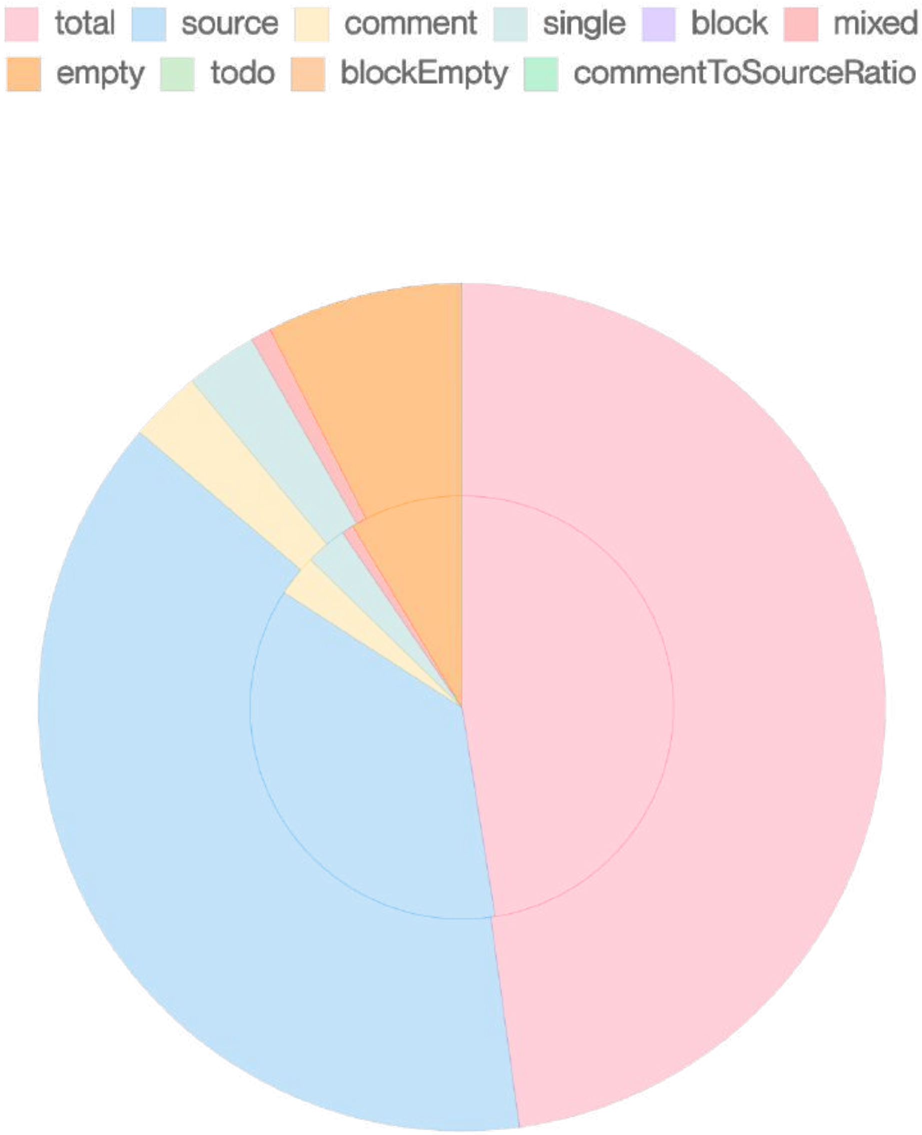
Issue Count



Risk Chart


perceivedComplexity
6.0
5.5
5.0
4.5
4.0
3.5
3.0
2.5
2.0
1.5
size
numLogicContracts
interfaceRisk
inlineDocumentation
compilerFeatures
compilerVersion
Average
Overall

Source Lines (sloc vs nsloc)



Inline Documentation

Comment-to-Source Ratio: On average there are 14 code lines per comment (lower=better)

ToDo's: 0

Risk represents the probability that a certain source-threat will exploit vulnerability, and the impact of that event on the organization or system. RiskLevel is computed based on CVSS version 3.0.

LEVEL	VALUE	VULNERABILITY	RISK
Critical	4	A vulnerability that can disrupt the contract functioning in a number of scenarios, or creates a risk that the contract may be broken.	Immediate action to reduce the risk level.
High	3	A vulnerability that affects the desired outcome when using a contract, or provides the opportunity to use a contract in an unintended way.	Implementation of corrective actions as soon as possible.
Medium	2	A vulnerability that could affects the desired outcome of executing the contract in a specific scenario.	Implementation of corrective actions in a certain period.
Low	1	A vulnerability that does not have a significant impact on possible scenarios for the use of the contract and is probably subjective.	Implementation of certain corrective actions or accepting the risk.
Informational	0	A vulnerability that have informational character but is not effecting any of the code.	An observation that does not determine a level of risk.

SCVE-0038 - MISSING NATSPEC DOCUMENTATION

Severity: Low
Files affected: MyToken.sol

DESCRIPTION	Solidity contracts can use a special form of comments to provide rich documentation for functions, return variables and more. This special form is named the Ethereum Natural Language Specification Format (NatSpec).
CODE	Line: 1 - MyToken.sol: pragma solidity ^0.7.0;
RECOMMENDATION	It is recommended to include natspec documentation and follow the doxygen style including @author, @title, @notice, @dev, @param, @return and make it easier to review and understand your smart contract.

SCVE-0002 - OUTDATED COMPILER VERSION

Severity: Low
Files affected: MyToken.sol

DESCRIPTION	Using an outdated compiler version can be problematic especially if there are publicly disclosed bugs and issues that affect the current compiler version.
CODE	Line: 1 - MyToken.sol: pragma solidity ^0.7.0;
RECOMMENDATION	It is recommended to use a recent version of the Solidity compiler.

SCVE-0003 - FLOATING PRAGMA

Severity: Low
Files affected: MyToken.sol

DESCRIPTION	The Contract should be deployed with the same compiler version and flags that they have been tested with thoroughly. Locking the pragma helps to ensure that contracts do not accidentally get deployed using, for example, an outdated compiler version that might introduce bugs that affect the contract system negatively.
CODE	Line: 1 - MyToken.sol: pragma solidity ^0.7.0;
RECOMMENDATION	Lock the pragma version and also consider known bugs (<a href="https://github.com/ethereum/solidity/releases">https://github.com/ethereum/solidity/releases</a>) for the compiler version that is chosen.

SCVE-0007 - AUTHORIZATION THROUGH TX.ORIGIN

Severity: Medium
Files affected: MyToken.sol

DESCRIPTION	tx.origin is a global variable in Solidity which returns the address of the account that sent the transaction. Using the variable for authorization could make a contract vulnerable if an authorized account calls into a malicious contract. A call could be made to the vulnerable contract that passes the authorization check since tx.origin returns the original sender of the transaction which in this case is the authorized account.

CODE	Line: 34 - MyToken.sol:
	owner = tx.origin;
	Line: 112 - MyToken.sol:
	require(tx.origin == owner, \"\");

RECOMMENDATION	tx.origin should not be used for authorization. Use msg.sender instead.

SCVE-0015 - ERC20 APPROVE

Severity: Low
Files affected: MyToken.sol

DESCRIPTION	The approve function of ERC-20 is vulnerable. Using front-running attack one can spend approved tokens before change of allowance value.
CODE	Line: 56 - MyToken.sol: function approve(
RECOMMENDATION	An attack vector on Approve.

SCVE-0037 - RACE CONDITIONS

Severity: High
Files affected: MyToken.sol

DESCRIPTION	TheERC20standard is quite well-known for building tokens on Ethereum. This standard has a potential frontrunning vulnerability which comes about due to theapprove()function. The standard specifies theapprove()function as: function approve(address _spender, uint256 _value) returns (bool success) This function allows a user to permit other users to transfer tokens on their behalf. The frontrunning vulnerability comes in the scenario when a user, Alice,approvesher friend,Bobto spend100 tokens. Alice later decides that she wants to revokeBob's approval to spend100 tokens, so she creates a transaction that setsBob's allocation to50 tokens.Bob, who has been carefully watching the chain, sees this transaction and builds a transaction of his own spending the100 tokens. He puts a highergasPriceon his transaction thanAlice's and gets his transaction prioritised over hers. Some implementations ofapprove()would allowBobto transfer his100 tokens, then whenAlice's transaction gets committed, resetsBob's approval to50 tokens, in effect givingBobaccess to150 tokens.
CODE	

	Line: 56 - MyToken.sol: function approve(
RECOMMENDATION	There are two classes of users who can perform these kinds of front-running attacks. Users (who modify thegasPriceof their transactions) and miners themselves (who can re-order the transactions in a block how they see fit). A contract that is vulnerable to the first class (users), is significantly worse-off than one vulnerable to the second (miners) as miner's can only perform the attack when they solve a block, which is unlikely for any individual miner targeting a specific block. Here I'll list a few mitigation measures with relation to which class of attackers they may prevent. One method that can be employed is to create logic in the contract that places an upper bound on thegasPrice. This prevents users from increasing thegasPriceand getting preferential transaction ordering beyond the upper-bound. This preventative measure only mitigates the first class of attackers (arbitrary users). Miners in this scenario can still attack the contract as they can order the transactions in their block however they like, regardless of gas price. A more robust method is to use acommit-revealscheme, whenever possible. Such a scheme dictates users send transactions with hidden information (typically a hash). After the transaction has been included in a block, the user sends a transaction revealing the data that was sent (the reveal phase). This method prevents both miners and users from frontrunning transactions as they cannot determine the contents of the transaction. This method however, cannot conceal the transaction value (which in some cases is the valuable information that needs to be hidden). TheENSsmart contract allowed users to send transactions, whose committed data included the amount of ether they were willing to spend. Users could then send transactions of arbitrary value. During the reveal phase, users were refunded the difference between the amount sent in the transaction and the amount they were willing to spend.

SCVE-0039 - PRIVATE HIDDEN VARIABLES

Severity: High
Files affected: MyToken.sol

DESCRIPTION	Private is the most restrictive visibility. State variables and functions marked as private are only visible and accessible in the same contract. Private is only a code-level visibility modifier. Your contract state marked as private is still visible to observers of the blockchain. It is just not accessible for other contracts.
CODE	Line: 14 - MyToken.sol: mapping(address => uint256) private _balances; Line: 15 - MyToken.sol:

	mapping(address => mapping(address => uint256)) private _allowances;
	
	Line: 17 - MyToken.sol:
	bytes32 private passwordHash;

RECOMMENDATION	Use private when you really want to protect your state variables and functions because you hide them behind logic executed through internal or public functions. Double check that the private modifier is not used to hide sensible information.

SCVE-0044 - INSUFFICIENT GAS GRIEFING

Severity: Medium
Files affected: MyToken.sol

DESCRIPTION	Insufficient gas griefing occurs when a smart contract function relies on an external contract call, and the caller can provide insufficient gas for the call to complete successfully. This can lead to denial of service and potential vulnerabilities in the contract logic.
CODE	Line: 76 - MyToken.sol: (bool success,) = msg.sender.call{ value: amount }(\"\");
RECOMMENDATION	Avoid using the .gas() function in external calls, as it may not provide enough gas for the called function to complete. Instead, use the .call{gas: _gas_amount}() syntax to ensure that the external call has sufficient gas. Additionally, use proper error handling and consider implementing a circuit breaker to halt contract execution if an error occurs.

SCVE-0046 - REENTRANCY ATTACK

Severity: Critical
Files affected: MyToken.sol

DESCRIPTION	Reentrancy attacks occur when a smart contract's external function call allows the called contract to call back into the original contract before the state variables have been updated. This can lead to unintended behavior and potentially allow an attacker to drain funds from the contract.

CODE	Line: 76 - MyToken.sol: (bool success,) = msg.sender.call{ value: amount }(\"\");
RECOMMENDATION	To prevent reentrancy attacks, use the 'checks-effects-interactions' pattern, ensuring that all state variables are updated before external calls. Consider using the OpenZeppelin ReentrancyGuard contract to mitigate reentrancy vulnerabilities.

SCVE-0051 - FRONT-RUNNING VULNERABILITY

Severity: Medium
Files affected: MyToken.sol

DESCRIPTION	Front-running vulnerabilities occur when a smart contract's functions are susceptible to manipulation by malicious users who monitor pending transactions and submit their own transactions with higher gas prices. This allows the attacker's transaction to be executed before the original transaction, potentially causing unintended consequences, such as manipulating the price of a token on a decentralized exchange.
CODE	Line: 76 - MyToken.sol: (bool success,) = msg.sender.call{ value: amount }(\"\");
RECOMMENDATION	To mitigate front-running vulnerabilities, consider implementing mechanisms such as commit-reveal schemes, batched transactions, or using a decentralized oracle to obtain external data securely. Additionally, design your contract functions to be resistant to front-running by reducing the potential benefits of front-running or making it more challenging to predict the outcome of a transaction.

SCVE-0052 - INCORRECT ACCESS CONTROL IMPLEMENTATION

Severity: Medium
Files affected: MyToken.sol

DESCRIPTION	Incorrect access control implementation can lead to vulnerabilities when a smart contract fails to properly restrict access to certain functions or state variables. This can result in unauthorized users being able to modify the contract's state or execute privileged functions.

CODE	Line: 37 - MyToken.sol:
	function balanceOf(address account) external view override returns (uint256) {
	Line: 74 - MyToken.sol:
	function withdraw(uint256 amount) external {
	Line: 106 - MyToken.sol:
	function getPasswordHash() external view returns (bytes32) {
	Line: 111 - MyToken.sol:
	function changeOwner(address newOwner, string memory newPassword) external {

RECOMMENDATION	Ensure that all sensitive functions and state variables have proper access control mechanisms in place, such as using modifiers like onlyOwner, onlyAdmin, or implementing custom access control logic. Consider using well-vetted and audited libraries, such as OpenZeppelin's AccessControl contract, to handle access control.

SCVE-0053 - LACK OF CONTRACT OWNERSHIP

Severity: Low
Files affected: MyToken.sol

DESCRIPTION	Lack of contract ownership can lead to unauthorized access and modification of contract state. When a contract lacks a clear ownership mechanism, there is no built-in access control for critical functions that modify the contract's state, making it more susceptible to unauthorized changes or exploitation. Establishing an ownership mechanism allows the contract owner or authorized entities to manage the contract, making it more secure and less prone to unauthorized modifications.

CODE	Line: 1 - MyToken.sol: pragma solidity ^0.7.0;
RECOMMENDATION	Implement an ownership mechanism, such as the Ownable pattern, to restrict access to sensitive functions. The Ownable pattern establishes an owner for the contract, typically assigning ownership to the contract deployer. To implement the Ownable pattern, create an owner state variable and an onlyOwner modifier. The onlyOwner modifier should be used to restrict access to critical functions that alter the contract's state or manage sensitive operations. Additionally, provide methods for transferring ownership and renouncing ownership to ensure proper management of the contract throughout its lifecycle. By implementing the Ownable pattern or other access control mechanisms, the contract will be more secure and less prone to unauthorized modifications.

Contract Summary

Report-Files Description Table

FILE NAME	SHA-1 HASH
/Code/solidcheck/backenMyToken.sol	6af4ef7e7eabb43fbd66afeb2f1d60f914fb1c2c

Contracts Description Table

CONTRACT	TYPE	BASES	
	FUNCTION NAME	VISIBILITY	MODIFIERS
VulnerableToken	Implementation	IERC20	
		Public	NO
	balanceOf	External	NO
	transfer	External	NO
	allowance	External	NO
	approve	External	NO
	transferFrom	External	NO
	withdraw	External	NO
	_transfer	Internal	
	_approve	Internal	
	getPasswordHash	External	NO
	changeOwner	External	NO