SKELETON ECOSYSTEM

SMART CONTRACT AUDIT

MEGALODON
[$MEGA]
ERC 20

0xC2B4eCcAFF699B4D18524bE3bCFFa28754421595

Table of Contents

Global Disclaimer

This document serves as a disclaimer for the crypto smart contract audit conducted by Skeleton Ecosystem. The purpose of the audit was to review the codebase of the smart contracts for potential vulnerabilities and issues. It is important to note the following:

Limited Scope: The audit is based on the code and information available up to the audit completion date. It does not cover external factors, system interactions, or changes made after the audit. The audit itself can not guarantee 100% safaty and can not detect common scam methods like farming and developer sell-out.

No Guarantee of Security: While we have taken reasonable steps to identify vulnerabilities, it is impossible to guarantee the complete absence of security risks or issues. The audit report provides an assessment of the contract’s security as of the audit date.

Continued Development: Smart contracts and blockchain technology are evolving fields. Updates, forks, or changes to the contract post-audit may introduce new risks that were not present during the audit.

Third-party Code: If the smart contract relies on third-party libraries or code, those components were not thoroughly audited unless explicitly stated. Security of these dependencies is the responsibility of their respective developers.

Non-Exhaustive Testing: The audit involved automated analysis, manual review, and testing under controlled conditions. It is possible that certain vulnerabilities or issues may not have been identified.

Risk Evaluation: The audit report includes a risk assessment for identified vulnerabilities. It is recommended that the development team carefully reviews and addresses these risks to mitigate potential exploits.

Not Financial Advice: This audit report is not intended as financial or investment advice. Decisions regarding the use, deployment, or investment in the smart contract should be made based on a comprehensive assessment of the associated risks.

By accessing and using this audit report, you acknowledge and agree to the limitations outlined above. Skeleton Ecosystem and its auditors shall not be held liable for any direct or indirect damages resulting from the use of the audit report or the smart contract itself.

Please consult with legal, technical, and financial professionals before making any decisions related to the smart contract.

Overview

Contract Name	Megalodon
Ticker/Simbol	$MEGA
Blockchain	Ethereum ERC20
Contract Address	0xC2B4eCcAFF699B4D18524bE3bCFFa28754421595
Creator Address	0xD40444F7cAaE402BB4830eCf4f69acA5a9ee1F4f
Current Owner Address	Renounced
Contract Explorer	https://etherscan.io/token/0xc2b4eccaff699b4d18524be3bcffa28754421595
Compiler Version	v0.8.19+commit.7dd6d404
License	MIT
Optimisation	Yes with 200 Runs
Total Supply	100,000,000 MEGA
Decimals	18

Creation/Audit

Contract Deployed	24 Aug 2023
Audit Created	09-Sept-23 20:00:00 UTC
Audit Update	V 0.1

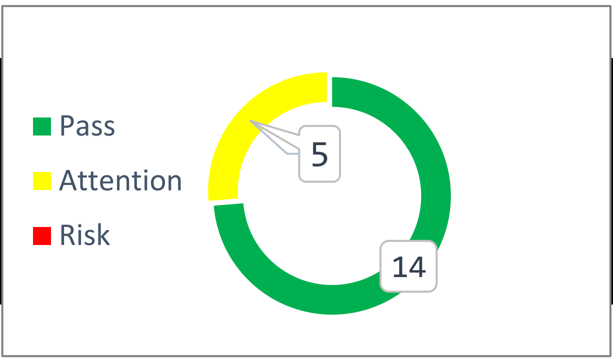
Verified Socials

Website	https://megaerc20.com
Telegram	https://t.me/MegaERC
Twitter (X)	https://www.twitter.com/megaerc20

MEGALODON ERC20

Contract Function Analysis

✓ Pass ⚠ Attention Item ❗ Risky Item



Contract Verified	✓	The contract source code is uploaded to blockchain explorer and is open source, so everybody can read it.
Contract Ownership	✓	The ownership of the contract was sent to dead address. With this the owner eliminates he's rights to modify the contract. The owner can not set any of the functions anymore.
Buy Tax	2 %	Shows the taxes for purchase transactions. Above 10% may be considered a high tax rate. More than 50% tax rate means may not be tradable. Contract renounced so tax rate is fixed.
Sell Tax	2 %	Shows the taxes for sell transactions. Above 10% may be considered a high tax rate. More than 50% tax rate means may not be tradable. Contract renounced so tax rate is fixed.
Honeypot Analyse	✓	Holder is able to buy and sell. If honeypot: The contract blocks sell transfer from holder wallet. Multiple events may cause honeypot. Trading disabled, extremely high tax
Liqudity Status	✓	Locked on 08.09.2023: 1: 86.17% Unicrypt for 169 days. 2: 13.79% Unicrypt for 169 days. Note! Initial liqudity tokens scanned. For new LP Lockers allways re-check with skeleton scanner on telegram.
Trading Disable Functions	⚠	Trading suspendable function found. If a suspendable code is included, the token maybe neither be bought or sold (honeypot risk). If contract is renounced this function can't be used. ✓ Renounced, this function can not be used.
Set Fees function	⚠	Fee Setting function found. The contract owner may contain the authority to modify the transaction tax. If the transaction tax is increased to more than 49%, the tokens may not be able to be traded (honeypot risk). If contract is renounced this function can't be used. ✓ Renounced, this function can not be used.
Proxy Contract	✓	Not a Proxy Contract. The proxy contract means contract owner can modifiy the function of the token and possibly effect the price. The Owner is not the creator but the creator may have authorisation to change functions.
Mint Function	✓	No mint function found. Mint function is transparent or non-existent. Hidden mint functions may increase the amount of tokens in circulation and effect the price of the token. Owner can mint new tokens and sell.

Balance Modifier Function	✅	No Balance Modifier function found. If there is a function for this, the contract owner can have the authority to modify the balance of tokens at other addresses. For example revoke the bought tokens from the holders wallet. Common form of scam: You buy the token, but it's disappearing from your wallet.
Blacklist Function	⚠️	Blacklist function found. If there is a blacklist, some addresses may not be able to trade normally. Example: you buy the token and right after your Wallet getting blacklisted. Like so you will be unable to sell. Honeypot Risk. ✅ Renounced, this function can not be used.
Whitelist Function	⚠️	Whitelist Function Found. If there is a function for this Developer can set zero fee or no max wallet size for adresses (for example team wallets can trade without fee. Can cause farming) ✅ Renounced, this function can not be used.
Hidden Owner Analysis	✅	No authorised hidden owner found. For contract with a hidden owner, developer can still manipulate the contract even if the ownership has been abandoned. Fake renounce.
Retrieve Ownership Function	✅	No functions found which can retrieve ownership of the contract. If this function exists, it is possible for the project owner to regain ownership even after relinquishing it. Also known as fake renounce.
Self Destruct Function	✅	No Self Destruct function found. If this function exists and is triggered, the contract will be destroyed, all functions will be unavailable, and all related assets will be erased.
Specific Tax Changing Function	⚠️	Specific Tax Changing Functions found. ✅ Renounced, this function can not be used. If it exists, the contract owner may set a very outrageous tax rate for assigned address to block it from trading. Can assign all wallets at once!
Trading Cooldown Function	✅	No Trading Cooldown Function found. If there is a trading cooldown function, the user will not be able to sell the token within a certain time or block after buying. Like a temporary honeypot.
Max Transaction and Holding Modify Function	✅	No Max Transaction and Holding Modify function found. ✅ Renounced, this function can not be used. If there is a function for this, the maximum trading amount or maximum position can be modified. Can cause honeypot
Transaction Limiting Function	✅	No Transaction Limiter Function Found. ✅ Renounced, this function can not be used. The number of overall token transactions may be limited (honeypot risk)

Contract Security

Total Findings: 7

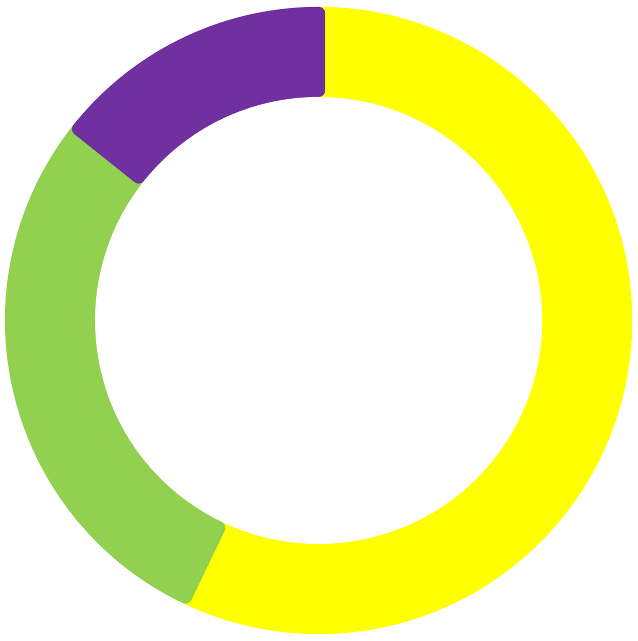
🟥 Critical 0

🟧 High 0

🟨 Medium 4

🟩 Low 2

🟪 Info 1



Contract Security

List of Found Issues

🟥 **No critical severity Issues found**

🟧 **No high severity issues found**

🟨 **Medium severity issues: (4)**

- Approve front running attack
- Unchecked Transfer
- Authorization through tx.origin
- Unchecked Array Lenght

🟩 **Low severity issues: (2)**

- Missing Events
- Long Number Literals

🟪 **Informational severity issues: (1)**

- Hard Coded Address

Contract Weakness Classisication

THE SMART CONTRACT WEAKNESS CLASSIFICATION REGISTRY (SWC REGISTRY) IS AN IMPLEMENTATION OF THE WEAKNESS CLASSIFICATION SCHEME PROPOSED IN EIP-1470. IT IS LOOSELY ALIGNED TO THE TERMINOLOGIES AND STRUCTURE USED IN THE COMMON WEAKNESS ENUMERATION (CWE) WHILE OVERLAYING A WIDE RANGE OF WEAKNESS VARIANTS THAT ARE SPECIFIC TO SMART CONTRACTS.

ID	Description	AI	Manual	Result
SWC-100	Function Default Visibility	Passed	Passed	Passed
SWC-101	Integer Overflow and Underflow	Passed	Passed	Passed
SWC-102	Outdated Compiler Version	Passed	Passed	Passed
SWC-103	Floating Pragma	Passed	Passed	Passed
SWC-104	Unchecked Call Return Value	Passed	Passed	Passed
SWC-105	Unprotected Ether Withdrawal	Passed	Passed	Passed
SWC-106	Unprotected SELFDESTRUCT Instruction	Passed	Passed	Passed
SWC-107	Reentrancy	Passed	Passed	Passed
SWC-108	State Variable Default Visibility	Passed	Passed	Passed
SWC-109	Uninitialized Storage Pointer	Passed	Passed	Passed
SWC-110	Assert Violation	Passed	Passed	Passed
SWC-111	Use of Deprecated Solidity Functions	Passed	Passed	Passed
SWC-112	Delegatecall to Untrusted Callee	Passed	Passed	Passed
SWC-113	DoS with Failed Call	Passed	Passed	Passed
SWC-114	Transaction Order Dependence	Passed	Passed	Passed
SWC-115	Authorization through tx.origin	low	medium	medium
SWC-116	Block values as a proxy for time	Passed	Passed	Passed
SWC-117	Signature Malleability	Passed	Passed	Passed
SWC-118	Incorrect Constructor Name	Passed	Passed	Passed
SWC-119	Shadowing State Variables	Passed	Passed	Passed
SWC-120	Weak Sources of Randomness from Chain Attributes	Passed	Passed	Passed

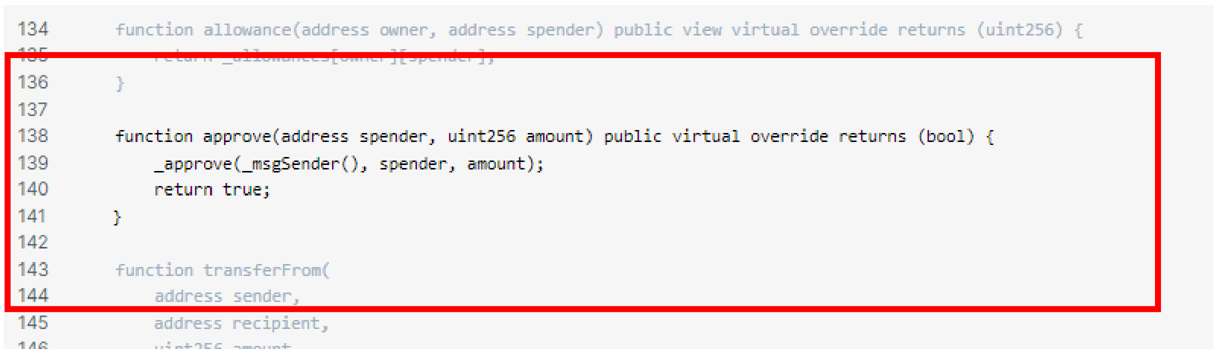
SWC-121	Missing Protection against Signature Replay Attacks	Passed	Passed	Passed
SWC-122	Lack of Proper Signature Verification	Passed	Passed	Passed
SWC-123	Requirement Violation	Passed	Passed	Passed
SWC-124	Write to Arbitrary Storage Location	Passed	Passed	Passed
SWC-125	Incorrect Inheritance Order	Passed	Passed	Passed
SWC-126	Insufficient Gas Griefing	Passed	Passed	Passed
SWC-127	Arbitrary Jump with Function Type Variable	Passed	Passed	Passed
SWC-128	DoS With Block Gas Limit	Passed	Passed	Passed
SWC-129	Typographical Error	Passed	Passed	Passed
SWC-130	Right-To-Left-Override control character (U+202E)	Passed	Passed	Passed
SWC-131	Presence of unused variables	Passed	Passed	Passed
SWC-132	Unexpected Ether balance	Passed	Passed	Passed
SWC-133	Hash Collisions With Multiple Variable Length Arguments	Passed	Passed	Passed
SWC-134	Message call with hardcoded gas amount	Passed	Passed	Passed
SWC-135	Code With No Effects	Passed	Passed	Passed
SWC-136	Unencrypted Private Data On-Chain	Passed	Passed	Passed

Detected Critical – High- Medium Severity Vulnerability Description

⚠ Approve Front Running Attack (2 Items)

Item: 1	Location: Line 138-141	Severity: 🟨 Medium

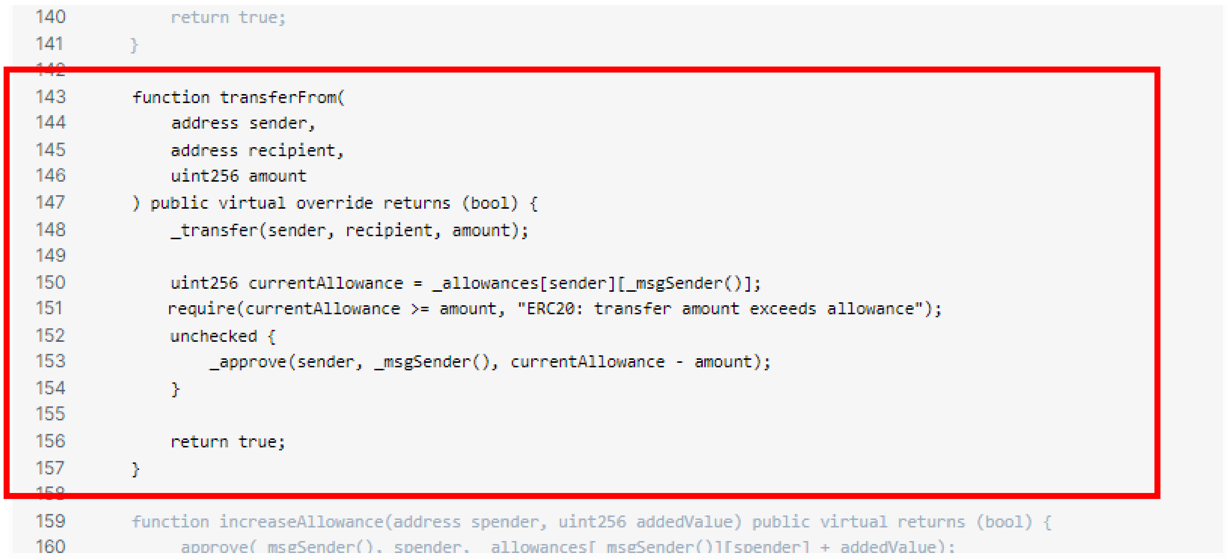
Function	The **approve()** method overrides current allowance regardless of whether the spender already used it or not, so there is no way to increase or decrease allowance by a certain value atomically unless the token owner is a smart contract, not an account. This can be abused by a token receiver when they try to withdraw certain tokens from the sender's account. Meanwhile, if the sender decides to change the amount and sends another approve transaction, the receiver can notice this transaction before it's mined and can extract tokens from both the transactions, therefore, ending up with tokens from both the transactions. This is a front-running attack affecting the ERC20 Approve function. The function approve can be front-run by abusing the **_approve** function.
Remedation	Only use the approve function of the ERC/BEP standard to change the allowed amount to 0 or from 0 (wait till transaction is mined and approved). Token owner just needs to make sure that the first transaction actually changed allowance from N to 0, i.e., that the spender didn't manage to transfer some of N allowed tokens before the first transaction was mined. Such checking is possible using advanced blockchain explorers such as [Etherscan.io](https://etherscan.io/) Another way to mitigate the threat is to approve token transfers only to smart contracts with verified source code that does not contain logic for performing attacks like described above, and to accounts owned by the people you may trust.



```
134        function allowance(address owner, address spender) public view virtual override returns (uint256) {
135            return _allowances[owner][spender];
136        }
137
138        function approve(address spender, uint256 amount) public virtual override returns (bool) {
139            _approve(_msgSender(), spender, amount);
140            return true;
141        }
142
143        function transferFrom(
144            address sender,
145            address recipient,
146            uint256 amount
```

Item: 2 Location: Line 143-157 Severity: 🟨 Medium

Function	The **transferFrom()** method overrides current allowance regardless of whether the spender already used it or not, so there is no way to increase or decrease allowance by a certain value atomically unless the token owner is a smart contract, not an account. This can be abused by a token receiver when they try to withdraw certain tokens from the sender's account. Meanwhile, if the sender decides to change the amount and sends another approve transaction, the receiver can notice this transaction before it's mined and can extract tokens from both the transactions, therefore, ending up with tokens from both the transactions. This is a front-running attack affecting the ERC20 Approve function. The function transferFrom can be front-run by abusing the **_approve** function.
Remedation	Only use the approve function of the ERC/BEP standard to change the allowed amount to 0 or from 0 (wait till transaction is mined and approved). Token owner just needs to make sure that the first transaction actually changed allowance from N to 0, i.e., that the spender didn't manage to transfer some of N allowed tokens before the first transaction was mined. Such checking is possible using advanced blockchain explorers such as [Etherscan.io](https://etherscan.io/) Another way to mitigate the threat is to approve token transfers only to smart contracts with verified source code that does not contain logic for performing attacks like described above, and to accounts owned by the people you may trust.

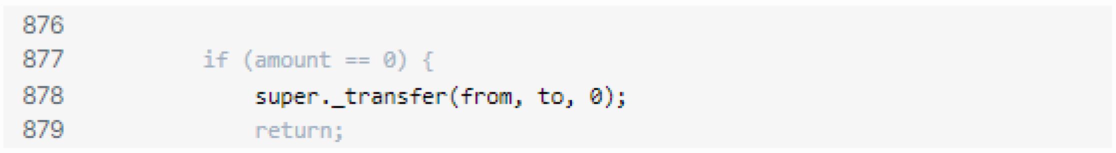


```
        return true;
    }

    function transferFrom(
        address sender,
        address recipient,
        uint256 amount
    ) public virtual override returns (bool) {
        _transfer(sender, recipient, amount);

        uint256 currentAllowance = _allowances[sender][_msgSender()];
        require(currentAllowance >= amount, "ERC20: transfer amount exceeds allowance");
        unchecked {
            _approve(sender, _msgSender(), currentAllowance - amount);
        }

        return true;
    }

    function increaseAllowance(address spender, uint256 addedValue) public virtual returns (bool) {
        _approve(_msgSender(), spender, _allowances[_msgSender()][spender] + addedValue);
```

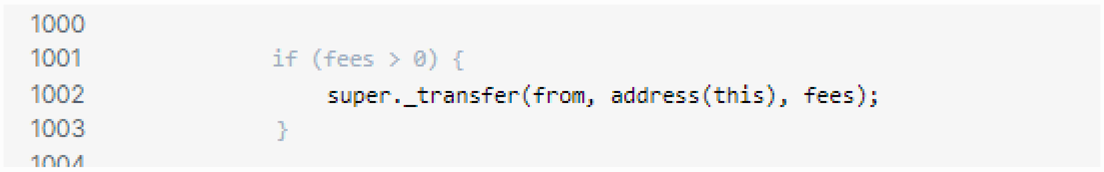
Unchecked Transfer (3 Items)

Item: 1	Location: Line 878	Severity:	🟨 Medium
Item: 2	Location: Line 1002	Severity:	🟨 Medium
Item: 3	Location: Line 1008	Severity:	🟨 Medium

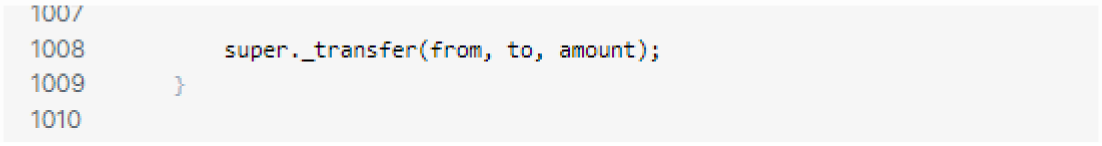
Function	Some tokens do not revert the transaction when the transfer or transferFrom fails and returns False. Hence we must check the return value after calling the transfer or transferFrom function.
Remedation	Use OpenZeppelin SafeERC20's safetransfer and safetransferFrom functions.



```
876
877        if (amount == 0) {
878            super._transfer(from, to, 0);
879            return;
```



```
1000
1001            if (fees > 0) {
1002                super._transfer(from, address(this), fees);
1003            }
1004
```

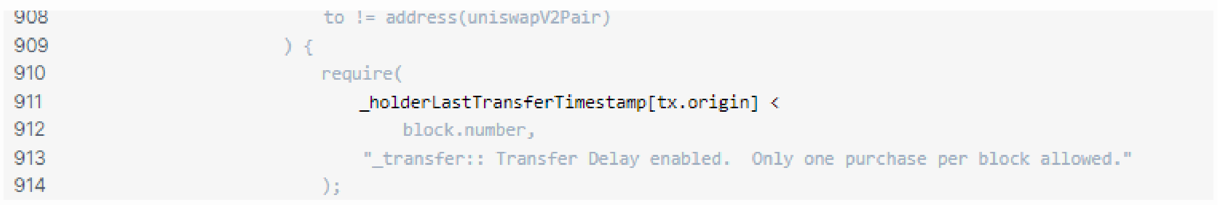


```
1007
1008        super._transfer(from, to, amount);
1009    }
1010
```

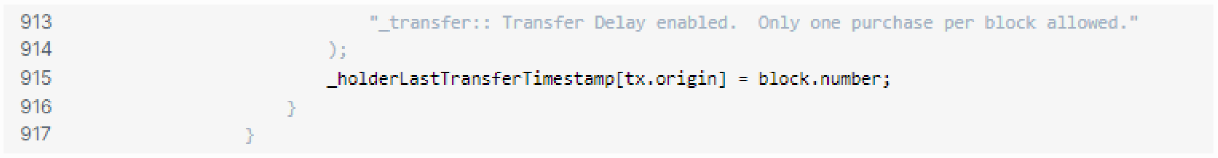
⚠ Authorization through tx.origin (2 Items)

Item: 1	Location:	Line 911	Severity:	🟨	Medium
Item: 2	Location:	Line 915	Severity:	🟨	Medium

Function	In Solidity, tx.origin is a global variable that returns the address of the account that sent the transaction. Using the variable for authorization could make a contract vulnerable. For example, if an authorized account calls a malicious contract which triggers it to call the vulnerable contract that passes an authorization check since tx.origin returns the original sender of the transaction which in this case is the authorized account.
Remedation	tx.origin should not be used for authorization in smart contracts. It does have some legitimate use cases, for example, To prevent external contracts from calling the current contract, you can implement a require of the form require(tx.origin == msg.sender). This prevents intermediate contracts from calling the current contract, thus limiting the contract to regular codeless addresses.



```
908                 to != address(uniswapV2Pair)
909             ) {
910                 require(
911                     _holderLastTransferTimestamp[tx.origin] <
912                         block.number,
913                     "_transfer:: Transfer Delay enabled.  Only one purchase per block allowed."
914                 );
```

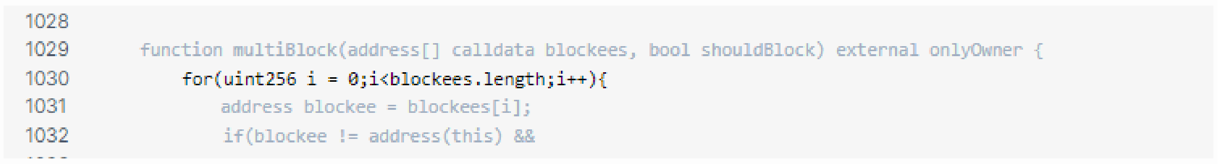


```
913                     "_transfer:: Transfer Delay enabled.  Only one purchase per block allowed."
914                 );
915                 _holderLastTransferTimestamp[tx.origin] = block.number;
916             }
917         }
```

⚠ Unchecked Array Lenght (1 Item)

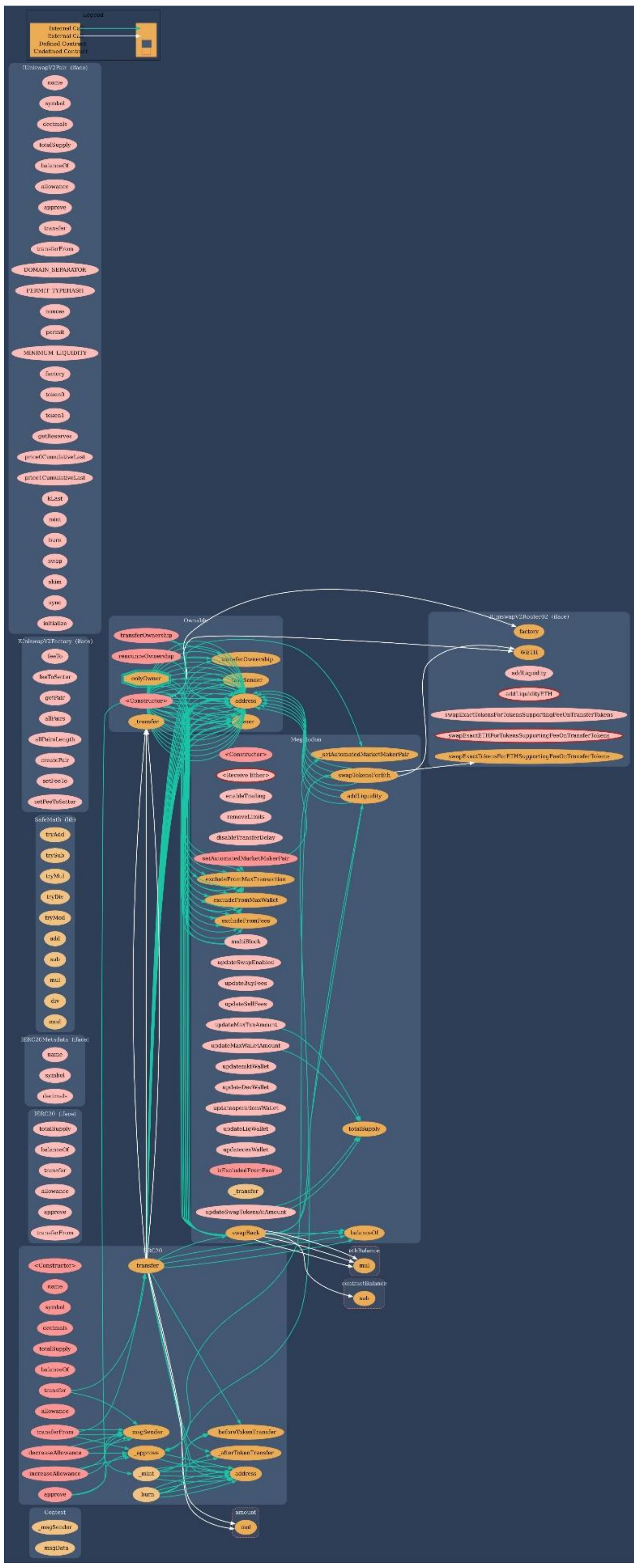
Item: 1	Location: Line 911	Severity: 🟨 Medium

Function	Ethereum is a very resource-constrained environment. Prices per computational step are orders of magnitude higher than with centralized providers. Moreover, Ethereum miners impose a limit on the total number of Gas consumed in a block. If array.length is large enough, the function exceeds the block gas limit, and transactions calling it will never be confirmed. `for (uint256 i = 0; i < array.length ; i++) { cosltyFunc(); }` This becomes a security issue if an external actor influences array.length. E.g., if an array enumerates all registered addresses, an adversary can register many addresses, causing the problem described above.
Remedation	Either explicitly or just due to normal operation, the number of iterations in a loop can grow beyond the block gas limit, which can cause the complete contract to be stalled at a certain point. Therefore, loops with a bigger or unknown number of steps should always be avoided.

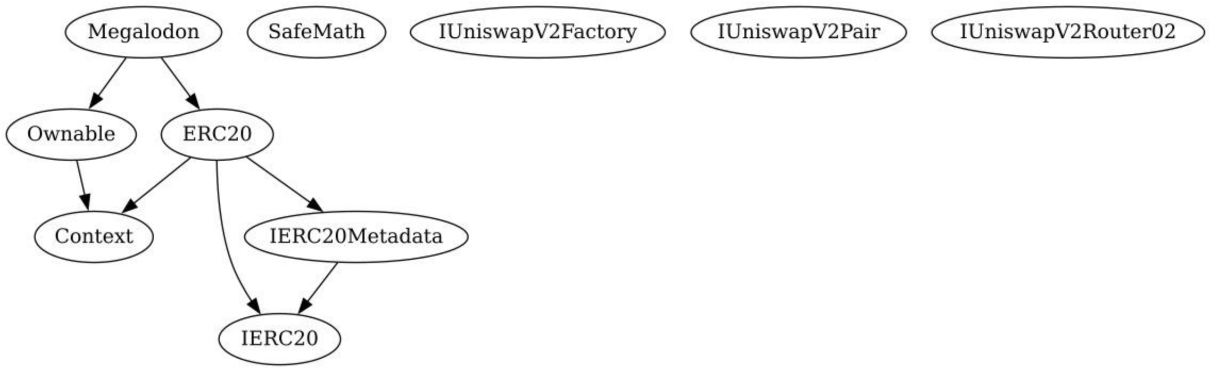


```
    function multiBlock(address[] calldata blockees, bool shouldBlock) external onlyOwner {
        for(uint256 i = 0;i<blockees.length;i++){
            address blockee = blockees[i];
            if(blockee != address(this) &&
```

Contract Flow Graph



Inheritance Graph



Contract Descriptions

Contract	**Type**	**Bases**		
	Function Name	**Visibility**	**Mutability**	**Modifiers**
Context	Implementation			
	_msgSender	Internal 🔒		
	_msgData	Internal 🔒		
Ownable	Implementation	Context		
		Public ❗	🛑	NO❗
	owner	Public ❗		NO❗
	renounceOwnership	Public ❗	🛑	onlyOwner
	transferOwnership	Public ❗	🛑	onlyOwner
	_transferOwnership	Internal 🔒	🛑	
IERC20	Interface			
	totalSupply	External ❗		NO❗
	balanceOf	External ❗		NO❗
	transfer	External ❗	🛑	NO❗
	allowance	External ❗		NO❗
	approve	External ❗	🛑	NO❗
	transferFrom	External ❗	🛑	NO❗
IERC20Metadata	Interface	IERC20		
	name	External ❗		NO❗
	symbol	External ❗		NO❗

	decimals	External ❗		NO❗
ERC20	Implementation	Context, IERC20, IERC20Metadata		
		Public ❗	🛑	NO❗
	name	Public ❗		NO❗
	symbol	Public ❗		NO❗
	decimals	Public ❗		NO❗
	totalSupply	Public ❗		NO❗
	balanceOf	Public ❗		NO❗
	transfer	Public ❗	🛑	NO❗
	allowance	Public ❗		NO❗
	approve	Public ❗	🛑	NO❗
	transferFrom	Public ❗	🛑	NO❗
	increaseAllowance	Public ❗	🛑	NO❗
	decreaseAllowance	Public ❗	🛑	NO❗
	_transfer	Internal 🔒	🛑	
	_mint	Internal 🔒	🛑	
	_burn	Internal 🔒	🛑	
	_approve	Internal 🔒	🛑	
	_beforeTokenTransfer	Internal 🔒	🛑	
	_afterTokenTransfer	Internal 🔒	🛑	
SafeMath	Library			
	tryAdd	Internal 🔒		
	trySub	Internal 🔒		
	tryMul	Internal 🔒		
	tryDiv	Internal 🔒		
	tryMod	Internal 🔒		
	add	Internal 🔒		
	sub	Internal 🔒		
	mul	Internal 🔒		

	div	Internal 🔒		
	mod	Internal 🔒		
	sub	Internal 🔒		
	div	Internal 🔒		
	mod	Internal 🔒		
IUniswapV2Factory	Interface			
	feeTo	External ❗		NO❗
	feeToSetter	External ❗		NO❗
	getPair	External ❗		NO❗
	allPairs	External ❗		NO❗
	allPairsLength	External ❗		NO❗
	createPair	External ❗	🛑	NO❗
	setFeeTo	External ❗	🛑	NO❗
	setFeeToSetter	External ❗	🛑	NO❗
IUniswapV2Pair	Interface			
	name	External ❗		NO❗
	symbol	External ❗		NO❗
	decimals	External ❗		NO❗
	totalSupply	External ❗		NO❗
	balanceOf	External ❗		NO❗
	allowance	External ❗		NO❗
	approve	External ❗	🛑	NO❗
	transfer	External ❗	🛑	NO❗
	transferFrom	External ❗	🛑	NO❗
	DOMAIN_SEPARATOR	External ❗		NO❗
	PERMIT_TYPEHASH	External ❗		NO❗
	nonces	External ❗		NO❗
	permit	External ❗	🛑	NO❗

	MINIMUM_LIQUIDITY	External ❗		NO❗
	factory	External ❗		NO❗
	token0	External ❗		NO❗
	token1	External ❗		NO❗
	getReserves	External ❗		NO❗
	price0CumulativeLast	External ❗		NO❗
	price1CumulativeLast	External ❗		NO❗
	kLast	External ❗		NO❗
	mint	External ❗	🛑	NO❗
	burn	External ❗	🛑	NO❗
	swap	External ❗	🛑	NO❗
	skim	External ❗	🛑	NO❗
	sync	External ❗	🛑	NO❗
	initialize	External ❗	🛑	NO❗
IUniswapV2Router02	Interface			
	factory	External ❗		NO❗
	WETH	External ❗		NO❗
	addLiquidity	External ❗	🛑	NO❗
	addLiquidityETH	External ❗	💵	NO❗
	swapExactTokensForTokensSupportingFeeOnTransferTokens	External ❗	🛑	NO❗
	swapExactETHForTokensSupportingFeeOnTransferTokens	External ❗	💵	NO❗

	swapExactTokensForETHSupportingFeeOnTransferTokens	External ❗	🛑	NO❗
Megalodon	Implementation	ERC20, Ownable		
		Public ❗	🛑	ERC20
		External ❗	💵	NO❗
	enableTrading	External ❗	🛑	onlyOwner
	removeLimits	External ❗	🛑	onlyOwner
	disableTransferDelay	External ❗	🛑	onlyOwner
	updateSwapTokensAtAmount	External ❗	🛑	onlyOwner
	updateMaxTxnAmount	External ❗	🛑	onlyOwner
	updateMaxWalletAmount	External ❗	🛑	onlyOwner
	excludeFromMaxTransaction	Public ❗	🛑	onlyOwner
	excludeFromMaxWallet	Public ❗	🛑	onlyOwner
	updateSwapEnabled	External ❗	🛑	onlyOwner
	updateBuyFees	External ❗	🛑	onlyOwner
	updateSellFees	External ❗	🛑	onlyOwner
	excludeFromFees	Public ❗	🛑	onlyOwner

	setAutomatedMarketMakerPair	Public ❗	🛑	onlyOwner
	_setAutomatedMarketMakerPair	Private 🔐	🛑	
	updatemktWallet	External ❗	🛑	onlyOwner
	updateDevWallet	External ❗	🛑	onlyOwner
	updateoperationsWallet	External ❗	🛑	onlyOwner
	updateLiqWallet	External ❗	🛑	onlyOwner
	updatecexWallet	External ❗	🛑	onlyOwner
	isExcludedFromFees	Public ❗		NO❗
	_transfer	Internal 🔒	🛑	
	swapTokensForEth	Private 🔐	🛑	
	multiBlock	External ❗	🛑	onlyOwner
	addLiquidity	Private 🔐	🛑	
	swapBack	Private 🔐	🛑	

🛑	Function can modify state	💵	Function is payable

Source:

```
File Name  SHA-1 Hash
c:\Solidity\megalodon.sol  160d67058d291bedaed417ca1ae6180d3f274800
```

Audit Scope

Audit Method.

Our smart contract audit is an extensive methodical examination and analysis of the smart contract’s code that is used to interact with the blockchain. Goal: discover errors, issues and security vulnaribilities in the code. Findings getting reported and improvements getting suggested.

Automatic and Manual Review
We are using automated tools to scan functions and weeknesses of the contract. Transfers, integer over-undeflow checks such as all CWE events.

Tools we use:
Visual Studio Code
CWE
SWC
Solidity Scan
SVD

In manual code review our auditor looking at source code and performing line by line examination. This method helps to clarify developer’s coding decisions and business logic.

Skeleton Ecosystem

https://skeletonecosystem.com

https://github.com/SkeletonEcosystem/Audits