BAILSEC.IO

OFFICE@BAILSEC.IO

X: @BAILSECURITY

TG: @HELLOATBAILSEC

FINAL REPORT

Prom

ERC20Classic

October 2024

Disclaimer:

Security assessment projects are time-boxed and often reliant on information that may be provided by a client, its affiliates, or its partners. As a result, the findings documented in this report should not be considered a comprehensive list of security issues, flaws, or defects in the target system or codebase.

The content of this assessment is not an investment. The information provided in this report is for general informational purposes only and is not intended as investment, legal, financial, regulatory, or tax advice. The report is based on a limited review of the materials and documentation provided at the time of the audit, and the audit results may not be complete or identify all possible vulnerabilities or issues. The audit is provided on an "as-is," "where-is," and "as-available" basis, and the use of blockchain technology is subject to unknown risks and flaws.

The audit does not constitute an endorsement of any particular project or team, and we make no warranties, expressed or implied, regarding the accuracy, reliability, completeness, or availability of the report, its content, or any associated services or products. We disclaim all warranties, including the implied warranties of merchantability, fitness for a particular purpose, and non-infringement.

We assume no responsibility for any product or service advertised or offered by a third party through the report, any open-source or third-party software, code, libraries, materials, or information linked to, called by, referenced by, or accessible through the report, its content, and the related services and products. We will not be liable for any loss or damages incurred as a result of the use or reliance on the audit report or the smart contract.

The contract owner is responsible for making their own decisions based on the audit report and should seek additional professional advice if needed. The audit firm or individual assumes no liability for any loss or damages incurred as a result of the use or reliance on the audit report or the smart contract. The contract owner agrees to indemnify and hold harmless the audit firm or individual from any and all claims, damages, expenses, or liabilities arising from the use or reliance on the audit report or the smart contract.

By engaging in a smart contract audit, the contract owner acknowledges and agrees to the terms of this disclaimer.

1. Project Details

Important:
Please ensure that the deployed contract matches the source-code of the last commit hash.

Project	PROM - ERC20Classic
Website	Prom.io
Language	Solidity
Methods	Manual Analysis
Github repository	https://github.com/prom-io/mymemes-app/tree/ef9e36f1e93f5f65a68b0e0f6c98b917d0dacb8b/packages/contracts/contracts/memify/core/erc20-classic
Resolution 1	https://github.com/prom-io/mymemes-app/tree/cb4fd48d9f7574275217faf93e5436090a160d09/packages/contracts/contracts/memify/core/erc20-classic

2. Detection Overview

Severity	Found	Resolved	Partially Resolved	Acknowledged (no change made)
High	2			2
Medium				
Low	4	1		3
Informational	10	8	1	1
Governance				
Total	16	9	1	6

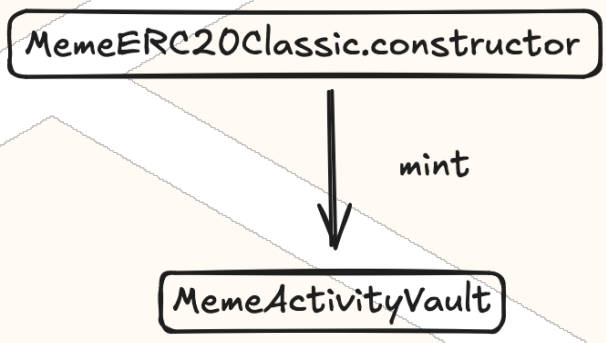
2.1 Detection Definitions

Severity	Description
High	The problem poses a significant threat to the confidentiality of a considerable number of users' sensitive data. It also has the potential to cause severe damage to the client's reputation or result in substantial financial losses for both the client and the affected users.
Medium	While medium level vulnerabilities may not be easy to exploit, they can still have a major impact on the execution of a smart contract. For instance, they may allow public access to critical functions, which could lead to serious consequences.
Low	Poses a very low-level risk to the project or users. Nevertheless the issue should be fixed immediately
Informational	Effects are small and do not post an immediate danger to the project or users
Governance	Governance privileges which can directly result in a loss of funds or other potential undesired behavior

3. Detection

MemeActivityVault

The MemeActivityVault contract is a simple custodian contract that receives and custodies the _activitityVaultAlloc which is minted during the MemeERC20Classic deployment.



The token owner can retrieve these tokens at any time via the retrieve function.

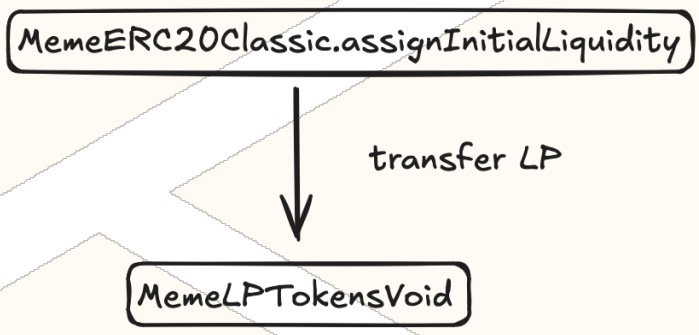
Privileged Functions

- retrieve

No issues found.

MemeLPTokensVoid

The MemeLPTokensVoid contract is a simple locker contract that receives LP tokens which are being permanently locked during the MemeERC20Classic.assignInitialLiquidity function. This contract has no functionality and thus all tokens sent to this contract remain permanently locked.



Privileged Functions

- none

No issues found.

MemeFactory

The MemeFactory contract is a factory contract that allows users to deploy new MemeERC20Classic tokens with custom names, symbols, and other parameters.
It ensures uniqueness by verifying that a token with the same name, symbol, and deployer doesn't already exist before deployment.
The contract uses the create2 opcode with a computed salt to deploy the new token at a deterministic address. Upon successful deployment, it records the new token's address and emits an event to log the creation.

Appendix: New deployment with salt

Each new MemeERC20Classic instance is deployed using the new keyword with a salt which is generated using the keccak256 hash function on the concatenation of the token's name, symbol, and the deployer's address. The new keyword combined with the salt is using the create2 opcode under the hood. This allows the contract address to be deterministically calculated before deployment. This predictability can be crucial for setting up contract interactions and dependencies in advance. Furthermore, due to the msg.sender inclusion, it prevents frontrunning of deployments from other users.

The contract address is computed based on four elements: a constant prefix, the deployer's address, the salt, and the bytecode hash of the contract.

address = keccak256(0xFF, deployer_address, salt, keccak256(contract_bytecode))

> 0xFF is a constant to prevent potential collisions.
> deployer_address is the address of the MemeFactory contract.
> salt is the unique value calculated earlier.
> keccak256(contract_bytecode) is the hash of the contract's creation code.

Appendix: Core Invariants

1. A deployer cannot deploy multiple tokens with the same name and symbol combination.
2. The token name and symbol must each be at least 3 characters long.
3. Every successfully deployed token is recorded in the tokenByName mapping and increments allTokensLength.
4. The address of the deployed MemeERC20Classic contract is deterministically generated using create2 with a salt that includes the deployer's address.

Privileged Functions

- none

Issue_01	Creation of fake tokens can trick buyers
Severity	**Low**
Description	The uniqueness enforcement of _name and _symbol is tied to the caller address. This means any user can spot the creation of a famous token and deploy a similar token with the exact same metadata, potentially tricking other users into buying it.
Recommendations	Consider if this will expose an issue, if yes, an idea could be to enforce that a _name and _symbol combination can be only deployed once. However, that would open up another DoS attack vector.
Comments / Resolution	Acknowledged.

Issue_02	Lack of address(0) check within constructor
Severity	**Informational**
Description	The constructor allows for the setting of multiple variables. Currently, there is no validation against address(0), which means in the scenario where one or many of these variables becomes address(0), the contract would not work. *address public v60Router;* *address public v60Factory;* *address public tokensVoid;* *address public memeActivityVault;*
Recommendations	Consider executing the aforementioned check.
Comments / Resolution	Resolved.

Issue_03	Immutable variables
Severity	**Informational**
Description	Variables which are defined within the constructor and never changed afterwards can be marked as immutable, this will include these variables directly into the contracts bytecode, which will save gas whenever these variables are accessed: *address public v60Router;* *address public v60Factory;* *address public tokensVoid;* *address public memeActivityVault;*
Recommendations	Consider marking these variables as immutable

Comments / Resolution	Resolved.

Issue_04	Unused variables
Severity	**Informational**
Description	Variables which are unused will unnecessarily increase the contract size for no reason and will confuse third-party reviewers: *address public voidVault;*
Recommendations	Consider removing all unused variables.
Comments / Resolution	Resolved.

Issue_05	Unused errors
Severity	**Informational**
Description	The contract contains one or more unused errors which will confuse third-party reviewers and increases the contract size for no reason: error InvalidDecimals(); error NotEnoughValue(); error InvalidLinksLength();
Recommendations	Consider removing all unused errors.
Comments / Resolution	Resolved.

MemeERC20Classic

The MemeERC20Classic contract is a custom ERC20 token which is meant to be deployed via the MemeFactory. Upon deployment, it mints the _memeOwnerAlloc amount to the deployer and _activityVaultAlloc amount to the MemeActivityVault contract. These amounts can be freely determined by the deployer.

The contract furthermore exposes an assignInitialLiquidity function which allows the owner to add the initial liquidity for the MEME/WETH ; WETH/MEME pair by providing a specific amount of ETH and minting a specific amount of MEME which is then added to the pair. The received LP token is then distributed as follows:

1) Owner
2) Locked
3) Burned

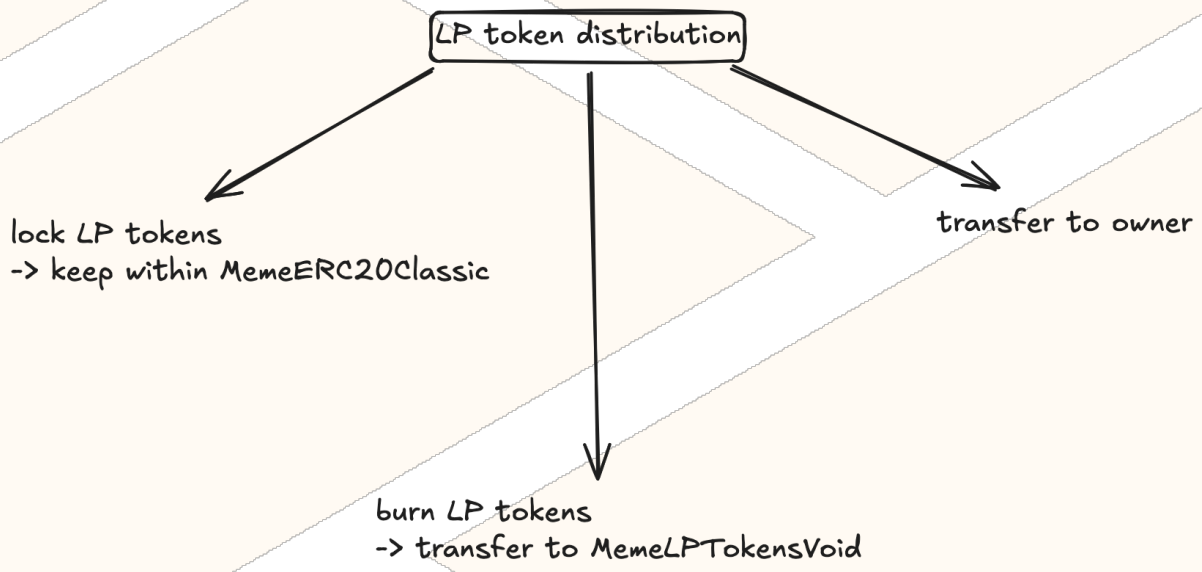
Appendix: LP token distribution

The minted LP token is distributed as follows:

a) percentageToBurn is transferred to the MemeLPTokensVoid contract, permanently removed from the supply.

b) percentageToLock is kept within the MemeERC20Classic contract and locked for a pre-determined time.

c) percentageToKeep is transferred to the owner.



Appendix: Core Invariants:

1. Tokens should never be distributed to users before liquidity has been added
2. Liquidity should never be added before the assignInitialLiquidity function is triggered
3. Locked LP tokens should only be claimable once endTimestamp is reached
4. Locked LP tokens should only be claimable once

Appendix: Locking Mechanism

Upon the initial LP creation, the percentageToLock amount is kept within the contract and locked for a determined time which is set by the owner as timeToLock. The endTimestamp is therefore set as block.timestamp + timeToLock and these tokens can be claimed via claimUnlockTokens towards the owner address whenever the endTimestamp is reached, permissionless.

Appendix: Initial Price Determination

The initial price of the UniswapV2Pair is determined during the execution of the assignInitialLiquidity function by the ratio of MEME tokens (initialTokenAmount) to ETH (msg.value) provided when adding liquidity.

Specifically, this function calls v60Router.addLiquidityETH, supplying both MEME tokens and ETH, where the amount of MEME tokens divided by the amount of ETH (or vice versa) sets the initial exchange rate in the liquidity pool. This ratio establishes the starting price at which MEME tokens can be traded for ETH (and vice versa) in the newly created liquidity pool immediately after deployment.

Suppose the owner calls assignInitialLiquidity with an initialTokenAmount of **100 MEME tokens** (represented as 100 * 10^18 in wei) and sends **10 ETH** (represented as 10 * 10^18 in wei) along with the transaction. By providing **100 MEME tokens** and **10 ETH** to the liquidity pool, the initial price is determined by their ratio:

Price of MEME in ETH: 0.1 ETH per MEME

This means that initially, each MEME token is valued at 0.1 ETH, or conversely, 1 ETH is worth 10 MEME tokens in the liquidity pool.

Privileged Functions

- assignInitialLiquidity

Issue_06	Malicious user can DoS liquidity addition
Severity	**High**
Description	During the assignInitialLiquidity function, the following check is executed: *if (_getV60Pair() != address(0)) revert PairExists();* This check is likely used to avoid a state where the pair has already liquidity added. However, any user can simply invoke the createPair function on the UniswapV2Factory contract, which is even possible without any owned tokens: *function createPair(* *address tokenA,* *address tokenB* *) external returns (address pair) {* *require(tokenA != tokenB, "UniswapV2: IDENTICAL_ADDRESSES");* *(address token0, address token1) = tokenA < tokenB* *? (tokenA, tokenB)* *: (tokenB, tokenA);* *require(token0 != address(0), "UniswapV2: ZERO_ADDRESS");* *require(* *getPair[token0][token1] == address(0),* *"UniswapV2: PAIR_EXISTS"* *); // single check is sufficient* *bytes memory bytecode = type(UniswapV2Pair).creationCode;* *bytes32 salt = keccak256(abi.encodePacked(token0, token1));* *assembly {* *pair := create2(0, add(bytecode, 32), mload(bytecode), salt)* *}* *IUniswapV2Pair(pair).initialize(token0, token1);* *getPair[token0][token1] = pair;* *getPair[token1][token0] = pair; // populate mapping in the*

	reverse direction *allPairs.push(pair);* *emit PairCreated(token0, token1, pair, allPairs.length);* *}* The assignInitialLiquidity function therefore always reverts.
Recommendations	Consider removing this check while simultaneously ensuring that no MemeERC20Classic tokens will be in circulation before the assignInitialLiquidity function is invoked. Furthermore, a check may be implemented which prevents the assignInitialLiquidity function from being called multiple times.
Comments / Resolution	Acknowledged, the AMM itself has been modified. Once Bailsec has audited the AMM and confirmed the implementation, this issue will be updated to resolved.

Issue_07	Tokens can remain stuck if liquidity has been added beforehand (when “Malicious user can DoS liquidity addition” is fixed)
Severity	**High**
Description	An important invariant of the contract is that no liquidity should be existent before the assignInitialLiquidity function is called. This is to prevent users from setting an arbitrary price and is enforced in the following check: *if (_getV60Pair() != address(0)) revert PairExists();* However, as we have mentioned, this check can be exploited to DoS the liquidity addition. The _addInitialLiq function exposes a slippage check:

	uint256 minEth = msg.value - (msg.value / 1000); *uint256 minToken = _tokenAmount - (_tokenAmount / 1000);* This slippage check is completely unnecessary if no initial liquidity has been added, as any desired ratio depending on the provided msg.value can be set, which means that always the full msg.value and _tokenAmount will be consumed. In such a scenario, tokens and ETH can remain permanently locked in the MemeERC20Classic contract if not the provided msg.value and initialTokenAmount is being added as liquidity due to the fact that the pair has a pre-existing ratio. If for example the provided msg.value is 10_000 ETH, it can happen that only 9990 ETH are being added to liquidity and the rest remains locked in the contract.
Recommendations	Consider ensuring that no tokens will be distributed before the initial liquidity is added which then ensures that no liquidity can be added. Additionally, once this is ensured, this slippage check can be removed.
Comments / Resolution	Acknowledged.

Issue_08	Arbitrary price creation can result in unexpected loss for initial buyers
Severity	**Low**
Description	The initial price for a token is trivially determined by the reserves within the UniswapV2Pair, which means that the owner can literally add any ratio. This may trick sniping bots into accepting any initial price.
Recommendations	We do not recommend a change as this would limit the price determination mechanism. However, this scenario should be kept in mind.
Comments / Resolution	Acknowledged.

Issue_09	Unnecessary complex amountToKeep calculation
Severity	**Low**
Description	The amountToKeep value is just what is left after amountToLock and amountToBurn has been deducted. Thus it can be trivially calculated as follows: amountToKeep = liqAmount - amountToLock - amountToBurn while ensuring that no dust remains in the contract. However, the current calculation is as follows: *amountToKeep = (percentageToKeep * liqAmount) / PERCENTAGE_DENOMINATOR;* *__transfer(* *address(this),* *memeOwnerAddress,* *amountToKeep + (liqAmount - amountToBurn - amountToLock - amountToKeep), // sending dust as well* *lpToken* *);* which is just unnecessary complex and will confuse users.
Recommendations	Consider simply using: liqAmount - amountToLock - amountToBurn
Comments / Resolution	Resolved.

Issue_10	_transfer function contains unused logic
Severity	**Low**
Description	The _transfer function contains the following logic: *function __transfer(* *address _from,* *address _to,* *uint256 _amount,* *address _paymentToken* *) internal {* *if (_paymentToken == address(0)) {* *if (_to != address(this)) {* *(bool success,) = payable(_to).call{value: _amount}(new bytes(0));* *require(success, 'Should transfer ethers');* *} else {* *require(msg.value >= _amount, 'not enough value');* *if (msg.value > _amount) {* *(bool success,) = payable(msg.sender).call{* *value: msg.value - _amount* *}(new bytes(0));* *require(success, 'Should transfer ethers');* *}* *}* *} else {* *if (_from == address(this)) {* *IERC20(_paymentToken).safeTransfer(_to, _amount);* *} else {* *IERC20(_paymentToken).safeTransferFrom(_from, _to, _amount);* *}* *}* *}*

	Only the red highlighted section is used, which makes all other functionality redundant and will only confuse third parties.
Recommendations	Consider simplifying the _transfer function to only handle the transfer out of the LP token. Optionally, the _transfer function can be completely removed and tokens can be transferred via a simple safeTransfer.
Comments / Resolution	Acknowledged.

Issue_11	Lack of parameter validation during assignInitialLiquidity
Severity	**Informational**
Description	The assignInitialLiquidity function allows the owner to add the initial liquidity. The owner can pass the following parameters: initialTokenAmount percentageToBurn percentageToLock timeToLock percentageToKeep as well as a msg.value The liquidity addition will always revert if either msg.value is zero or if initialTokenAmount is zero: *if (_totalSupply == 0) {* *liquidity = Math.sqrt(amount0.mul(amount1)).sub(MINIMUM_LIQUIDITY);* *_mint(address(0), MINIMUM_LIQUIDITY); // permanently lock the first MINIMUM_LIQUIDITY tokens* *}*
Recommendations	Consider validating both parameters accordingly.
Comments / Resolution	Resolved.

Issue_12	block.timestamp increase is pointless
Severity	**Informational**
Description	The _addInitialLiq function interacts with the router as follows: *(, , uint256 liquidityAmount) = v60Router.addLiquidityETH{value: msg.value}(* *address(this),* *_tokenAmount,* *minToken,* *minEth,* *address(this),* *block.timestamp + 20 minutes* *);* The increase of block.timestamp by 20 minutes is redundant as the block.timestamp will not change whenever a transaction is being executed. Furthermore, it is not necessary to pass a custom deadline as parameter because there will not be any slippage, which means it is irrelevant if the transaction remains stuck in the mempool.
Recommendations	Consider simply using block.timestamp.
Comments / Resolution	Acknowledged.

Issue_13	Unused events
Severity	**Informational**
Description	Certain events are unused, this does not only confuse third party reviewers but will also affect other parties which rely on the correctness of event emission. *event LatestPrice(uint256 amount);* *event MemeOwnerAirdrop(uint amount);* *event CoinCreation(* *string name,* *string symbol,* *address wprom,* *uint256 memeOwnerAlloc,* *address celebrityAddress* *);*
Recommendations	Consider removing all unused events.
Comments / Resolution	Partially resolved.

Issue_14	Unused variables
Severity	**Informational**
Description	Variables which are unused will unnecessarily increase the contract size for no reason and will confuse third-party reviewers. *uint256 public claimedOwnerShare;*
Recommendations	Consider removing all unused variables.
Comments / Resolution	Resolved.

Issue_15	ERC165 interface is not extended
Severity	**Informational**
Description	Generally speaking, the ERC165 interface is meant to be overridden and extended with a custom interface Id: */** * @dev Implementation of the {IERC165} interface. * * Contracts that want to implement ERC-165 should inherit from this contract and override {supportsInterface} to check * for the additional interface id that will be supported. For example: * * ```solidity * function supportsInterface(bytes4 interfaceId) public view virtual override returns (bool) { * return interfaceId == type(MyInterface).interfaceId || super.supportsInterface(interfaceId); * } * ```*

	*/ This is not happening here.
Recommendations	Consider if it is desired to expose a custom interface.
Comments / Resolution	Resolved, this has been removed.

Issue_16	Unused errors
Severity	**Informational**
Description	The contract contains one or more unused errors which will confuse third-party reviewers and increases the contract size for no reason: error NotEnoughValue(); error AllocIsTooBig();
Recommendations	Consider removing all unused errors.
Comments / Resolution	Resolved.