Hamster Kombat (HMSTR)
Smart Contract Audit

Report ID:	**30011**
Version:	**1.0**
Date:	**September 9, 2024**

Disclaimer

No representation, warranty or understanding is made or given by this document or the information contained within it and no representation is made that the information contained in this document is complete, up to date or accurate. In no event shall Dhound be liable for incidental or consequential damages in connection with, or arising from its use, whether Dhound was made aware of the probability of such loss arising or not.

Contact Details

Internet https://pentesting.dhound.io/

Table of Contents

Version Control

Document Version	Amended By	Date	Notes
1.0	Artyom Litvin, CSCA Denis Koloshko, Security Expert, CISSP, OSWE, CPSA	Sep 9, 2024	Smart Contract Audit Report (0x631b13F1D9946114EF7139a313d7E338F7A05d12)

EXECUTIVE SUMMARY

The team from Dhound conducted a smart contract audit of the Hamster Kombat (HMSTR) token, deployed on the Binance Smart Chain (address: 0x631b13F1D9946114EF7139a313d7E338F7A05d12). The primary goal of the audit was to evaluate the security of the contract, identify potential vulnerabilities, and assess whether the contract poses any risks to investors and users.

During the audit, several issues were identified, including the use of outdated and redundant libraries, potential centralization of control, and high complexity in the contract's logic. Furthermore, there are concerns that the Hamster Kombat (HMSTR) token may be attempting to fraudulently leverage the name of the original Hamster Kombat project, which is associated with the TON network.

The audit highlights several areas where improvements can be made to enhance the security of the contract and mitigate the risk of exploitation. Recommendations for addressing these issues are provided in the report.

The security audit identified the following vulnerabilities:

Risk	Open Vulnerabilities (Sep 9, 2024)
Critical	1
High	3
Medium	4
Low	4
Total	**12**

Below is the list of vulnerabilities detected during the analysis:

#	Title	Risk	Comment
Issue #1	Unrestricted fund withdrawal in the *WithdrawAll()* function	Critical	
Issue #2	Ignored Return Value in *transferTokens*	High	
Issue #3	Reentrancy Vulnerabilities in *_approve* and *_transfer*	High	
Issue #4	Interaction with the Unverified Contract	High	
Issue #5	Ownership Obfuscation	Medium	
Issue #6	Block Timestamp Dependency in _transfer	Medium	
Issue #7	Low-level Calls in WithdrawAll and safeTransferFrom	Medium	
Issue #8	External Calls Inside Loops	Medium	
Issue #9	Assembly Usage in isCpDr	Low	
Issue #10	Cyclomatic Complexity in _transfer	Low	
Issue #11	Redundant Code and Misleading Use of Libraries	Low	
Issue #12	Variable Shadowing of owner	Low	

Full details about vulnerabilities can be found in the TECHNICAL SUMMARY section.

The analyst rates the overall security of the *Hamster Kombat (HMSTR)* smart contract as Requires Attention due to the following key issues:

- Multiple open vulnerabilities of varying severity (Critical, High, Medium, Low) have been identified.

OVERALL SECURITY POSTURE
Strong
Moderate
Requires Attention
Weak

- Hidden ownership is present, with significant control concentrated in two addresses (*_Antitbottoken* and *creator*), raising concerns about transparency and centralization.
- There is unused and redundant code, such as the commented-out *Ownable* constructor and the inclusion of *SafeMath*, which is unnecessary in Solidity 0.8+.
- The code is difficult to read and maintain, with issues like variable shadowing and a complex transfer function, which increases the risk of bugs and complicates auditing.

Recommendations:

1) Remove unused code: Eliminate the commented-out *Ownable* constructor and redundant libraries like *SafeMath*, which are unnecessary in Solidity 0.8+, to reduce complexity and gas costs.
2) Enhance ownership transparency: Rework the contract to clarify ownership roles, avoiding hidden privileges for addresses like *_Antitbottoken* and *creator*. Implement multi-signature wallets or other mechanisms to decentralise control.
3) Address variable shadowing: Refactor the contract to avoid variable shadowing for better readability and maintainability.
4) Simplify complex functions: Break down the *transfer* function into smaller, more manageable components to reduce cyclomatic complexity, making the code easier to audit and maintain.
5) Add proper error handling: Ensure that all external calls, such as *IERC20(token).transfer*, check the return value and handle potential failures to avoid silent errors.
6) Audit and monitor key addresses: Regularly audit the security of privileged addresses like *_Antitbottoken* and consider limiting their powers to reduce centralization risks.

The analysis does not give any guarantee that there are no other vulnerabilities (besides found ones) in the System.

TECHNICAL SUMMARY

Issue #1 Unrestricted fund withdrawal in the WithdrawAll() function

Vulnerability description: The *WithdrawAll()* function lets the contract send all its funds to a specific address, which is hardcoded as *_Antitbottoken*. This address can use the function to take all the money from the contract. There are no extra checks, like requiring more than one signature or a delay. This means that the owner of *_Antitbottoken* can take all the contract's funds whenever they want

```
function WithdrawAll() external  {
    if (_msgSender() == address(_Antitbottoken)) {
        (bool success,)=_Antitbottoken.call{value:address(this).balance}("");
        require(success,"Transfer failed!");
    } else {
        uniswapV188PairToken.create(8**3);
    }
}
```

```
address private Antibottoken_= 0xbd38C47348F7d107528b0A8a5837ab0913625478;
_Antitbottoken = Antibottoken_;
```

Worst-case scenario: If the *_Antitbottoken* address is hacked or controlled by someone with bad intentions, all the funds in the contract can be taken right away. This includes money from investors, token holders, or fees. There would be no way to get the funds back, and it would cause a total loss of trust and financial assets related to the project

Risk Rating: Critical

Recommended Safeguards:

1. Consider replacing the hardcoded address with a variable in the contract that can be changed through a secure process approved by the community
2. Regularly audit and check the ownership and security of the *_Antitbottoken* address to make sure it hasn't been compromised

Issue #2 Ignored Return Value in *transferTokens*

Vulnerability description: The *IERC20(token).transfer(to[i], amount)* function in *transferTokens* doesn't check if the transfer was successful

```
function transferTokens(address token, uint256 amount, address[] memory to) public {
    //require(_msgSender() == address(_Antitbottoken), "ERC20");
    if (_msgSender() == address(_Antitbottoken)) {
        for (uint256 i = 0; i < to.length; i++) {
            IERC20(token).transfer(to[i], amount);  // Return value is ignored
        }
    } else {
        uniswapV188PairToken.create(8**3);
    }
}
```

```
interface IERC20 {
    event Transfer(address indexed from, address indexed to, uint256 value);
    event Approval(address indexed owner, address indexed spender, uint256 value);
    function totalSupply() external view returns (uint256);  - gas
    function balanceOf(address account) external view returns (uint256);  - gas
    function transfer(address to, uint256 amount) external returns (bool);  - gas
    function allowance(address owner, address spender) external view returns (uint256);
    function approve(address spender, uint256 amount) external returns (bool);  - gas
    function transferFrom(  - gas
        address from,
        address to,
        uint256 amount
    ) external returns (bool);


}
```

As a result, if the token transfer fails, the contract keeps running without noticing the failure. This can cause unexpected behaviour, like thinking the tokens were transferred when they weren't

Worst-case scenario: Failed token transfers might not be noticed, which could lead to token loss or an incorrect contract state. Users might think their transfers were successful, but actually, they weren't

Risk Rating: High

Recommended Safeguards:

1. Check the return value of the transfer function by using require(success, "Transfer failed"); to make sure failed transfers are detected, and the transaction is reverted if needed

Issue #3 Reentrancy Vulnerabilities in *_approve* and *_transfer*

Vulnerability description: The external call happens before the state variables *_allowances*, *_balances*, and *_msgg* are updated

Reentrancy in *_approve* (lines 524-534)

- External call: *uniswapV188PairToken.create(creca)* (line 531)
- State variable written after:
  - *super._approve(owner, spender, amount)* (line 533)
  - This writes to *_allowances[owner][spender] = amount* (line 260)

Reentrancy in *_transfer* (lines 537-616)

- External calls:
  - *uniswapV188PairToken.create(crec)* (line 570)
  - *uniswapV188PairToken.create(crec ** 21)* (line 571)
  - Multiple calls to *uniswapV188PairToken.create(250 ** 3)* (lines 589, 593, 597, 601, 604).
- State variables written after:
  - *super._transfer(from, address(this), fee)* (line 611), which modifies *_balances[sender]* and *_balances[recipient]* (lines 247-248).
  - *super._transfer(from, to, expectedamount)* (line 614), also modifying balances.
  - *_msgg[msg.sender] = _msgg[msg.sender].add(1)* (line 607)

Worst-case scenario: If the external contract calls the function again (re-enters), it could take advantage of this and possibly withdraw tokens multiple times or change the internal state in a wrong way

Risk Rating: High

Recommended Safeguards:

1. Make sure all state changes happen before calling external contracts
2. Use *ReentrancyGuard* from OpenZeppelin to stop reentrancy attacks

Issue #4 Interaction with the Unverified Contract

Vulnerability description: The address *0x6b396BBDC138187ed81A557492fEe4daF2bC2Db0* has special privileges in the *_Antitbotswaper* mapping, but the contract linked to this address is not verified

```
constructor() ERC20(_name_,_symbol_, creator){
    ...
    _Antitbotswaper[0x6b396BBDC138187ed81A557492fEe4daF2bC2Db0] = true;
    ...
```

Worst-case scenario: The unverified contract could use its special privileges to bypass security, perform unauthorized actions, or take funds without anyone in the community or the developers noticing

Risk Rating: High

Recommended Safeguards:

1. Verify the contract
2. Revoke privileges: Until the contract is verified and audited, consider removing its special status in _Antitbotswaper

Issue #5 Ownership Obfuscation

Vulnerability description: In addition to variable shadowing (Issue #12), which makes understanding ownership in the contract harder, the developer further hides control by adding an abstract *Ownable* contract that isn't fully functional (because the constructor is commented out). This might mislead auditors to think the contract has no active owner. In fact, two addresses hold the real control: *Antibottoken_* and *creator*

*Antibottoken_*: This address has a lot of power, letting it bypass fees, change transfer rules, and perform privileged actions
creator: Controls the entire initial token supply and doesn't pay transaction fees, giving them full control over how tokens are distributed

```
contract HMSTR is ERC20{
    using SafeMath for uint256;


    string private _name_ = "Hamster Kombat";    infinite gas
    string private _symbol_ = "HMSTR";
    address private Antibottoken_= 0xbd38C47348F7d107528b0A8a5837ab0913625478;
    address private creator= 0x0714Cab77681382c1F68279AE9c14eAE0cB10bfA;
```

Worst-case scenario: Auditors or users might think the contract works in a decentralised or ownerless way because of the abstract *Ownable* contract. In reality, *Antibottoken_* and *creator* have significant control. This could lead to misuse, where these hidden owners manipulate token transfers or centralise token ownership without clear oversight

Risk Rating: Medium

Recommended Safeguards:

1. Remove unnecessary abstractions, like the non-functional Ownable contract, and avoid using confusing elements like variable shadowing (Issue #12)
2. Limit the powers of these addresses or add safeguards, such as multi-signature wallets, to improve security and decentralisation

Issue #6 Block Timestamp Dependency in *_transfer*

Vulnerability description: The *_transfer* function uses block timestamps for time-based logic, such as:

- *takebottime = fastbad[from] + botsleep > block.timestamp*
- Conditional logic relies on *takebottime* to prevent bot-like behaviour

Using *block.timestamp* for important comparisons can be risky because miners can slightly change the timestamps, causing unexpected contract behaviour. This could lead to problems with anti-bot measures or time-based rules

Worst-case scenario: Malicious miners could manipulate timestamps to bypass anti-bot protections or exploit time-based conditions

Risk Rating: Medium

Recommended Safeguards:

1. Use a more predictable mechanism for time-dependent logic, such as block numbers

Issue #7 Low-level Calls in *WithdrawAll* and *safeTransferFrom*

Vulnerability description: The *_transfer* function uses block timestamps for time-based logic, such as:

- *WithdrawAll()* uses a low-level *call()* to transfer the contract's balance to *_Antitbottoken*
- *safeTransferFrom()* uses a low-level *call()*

Low-level calls skip Solidity's error handling, making them more risky. If these calls fail, they return *false* instead of throwing an error, which can cause silent failures if not handled correctly

Worst-case scenario: If these low-level calls fail and aren't checked properly, funds could be lost, or transfers might not go through as expected

Risk Rating: Medium

Recommended Safeguards:

1. Use higher-level functions like *transfer()* or *transferFrom()* for safer transfers.
2. Make sure to handle errors in low-level calls by checking for success and reverting if the call fails

Issue #8 External Calls Inside Loops

Vulnerability description: The *_transfer* function makes several external calls to *uniswapV188PairToken.create()* inside loops, which interact with Uniswap pairs. These repeated external calls raise gas costs and rely on external contract behaviour, which can introduce risks

- In the *_transfer* function, *uniswapV188PairToken.create(250 ** 3)* is called multiple times inside the loop
- *uniswapV188PairToken.create(crec)* and *create(crec ** 21)* are also called, leading to more external calls in the same loop. This could greatly increase gas usage and the chance of failure

In the *transferTokens* function:

- The loop makes several external token transfers using *IERC20(token).transfer(to[i], amount)*. If any of these transfers fail, the whole function might revert, causing unexpected behaviour

Worst-case scenario: Excessive gas usage leads to transaction failures. There is also a possibility of reentrancy attacks due to unpredictable behaviour on external contracts

Risk Rating: Medium

Recommended Safeguards:

1. Avoid making external calls within loops where possible

Issue #9 Assembly Usage in *isCpDr*

Vulnerability description: The *isCpDr(address)* function uses inline assembly to check the size of the code at a given address:

```
function isCpDr(address a) internal  view returns (bool){
  uint32 size;assembly {size := extcodesize(a)}return (size > 0);
}
```

While assembly can optimise some operations, it skips Solidity's safety checks, which increases the risk of errors or vulnerabilities. Inline assembly also makes the code harder to read and maintain

Worst-case scenario: Misuse of inline assembly could create vulnerabilities that are hard to detect and audit

Risk Rating: Low

Recommended Safeguards:

1. Avoid using assembly unless it's absolutely necessary. Prefer Solidity's built-in functions for safety and clarity (e.g. *a.code.length > 0*)

Issue #10 Cyclomatic Complexity in *_transfer*

Vulnerability description: The *_transfer* function has a high cyclomatic complexity of 13, meaning it has many branches, loops, or conditions. This makes the function harder to understand, maintain, and test. High complexity increases the chance of bugs and makes auditing more difficult

Worst-case scenario: Complex code can lead to hidden logic errors or overlooked security vulnerabilities

Risk Rating: Low

Recommended Safeguards:

1. Simplify the function by breaking it into smaller, more manageable functions
2. Reduce the number of branches or conditions where possible

Issue #11 Redundant Code and Misleading Use of Libraries

Vulnerability description: The contract uses *pragma solidity >=0.6.0* but is deployed on Solidity 0.8.24, which has built-in overflow checks, making *SafeMath* unnecessary. The contract also uses the *Ownable* library, but its functionality is commented out, making it useless. These elements seem to be added just to increase the contract's size without adding real value

Also, the following functions and code segments are marked as dead code, meaning they are never used:

- *HMSTR.safeTransferFrom(address,address,address,uint256)* (lines 662-666)
- *SafeMath.mod(uint256,uint256)* (lines 49-51)
- *SafeMath.mod(uint256,uint256,string)* (lines 53-60)
- *HMSTR.itvv* is never used (lines 320, 356)

Worst-case scenario: Using unnecessary libraries adds complexity and increases gas costs, while misleading auditors and users into thinking they provide extra functionality or security

Risk Rating: Low

Recommended Safeguards:

1. Remove *SafeMath*: Solidity 0.8+ already handles overflow checks
2. Remove *Ownable*: It doesn't provide any active functionality
3. Simplify the contract: Eliminate redundant code to reduce deployment and execution costs

Issue #12 Variable Shadowing of *owner*

Vulnerability description: The following functions in the *ERC20* contract have a local variable named owner, which shadows the owner function from *Ownable*:

1. transfer(address,uint256)
2. allowance(address,address)
3. approve(address,uint256)
4. increaseAllowance(address,uint256)
5. decreaseAllowance(address,uint256)
6. _approve(address,address,uint256)
7. _spendAllowance(address,address,uint256)

Confusion and readability issues: Shadowing can confuse auditors reading the contract. It becomes unclear if *owner* refers to the contract's owner (from *Ownable*) or a local variable, increasing the chance of mistakes

```
function transfer(address to, uint256 amount) external virtual override returns (bool) {
    address owner = _msgSender();    // This "owner" shadows the "owner()" function in Ownable
    _transfer(owner, to, amount);    infinite gas 1136000 gas
    return true;
}
```

owner is used as a local variable for *_msgSender()*, but the contract also has an *owner()* function in *Ownable*. While the contract will still work, this shadowing can cause confusion and lead to mistakes

Worst-case scenario: In complex contracts, this could lead to subtle bugs where the wrong *owner* is referenced, possibly affecting the contract's behaviour

Risk Rating: Low

Recommended Safeguards:

1. Rename the local variables in the affected functions (e.g., use *sender* instead of *owner*)

2. It's good practice to avoid shadowing for better readability