Cyberscope

Audit Report

Flying Avocado Cat

October 2024

Network ETH

Address 0x1a3A8Cf347b2bF5890D3D6A1B981c4f4432C8661

Audited by © cyberscope

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	FLYINGAVOCADOCAT
Compiler Version	v0.8.22+commit.4fc1097e
Optimization	200 runs
Explorer	https://etherscan.io/address/0x1a3a8cf347b2bf5890d3d6a1b981c4f4432c8661
Address	0x1a3a8cf347b2bf5890d3d6a1b981c4f4432c8661
Network	ETH
Symbol	FAC
Decimals	18
Total Supply	10,000,000

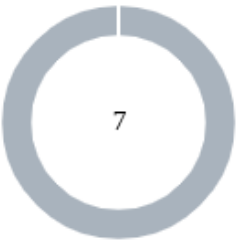
Audit Updates

Initial Audit	11 Oct 2024

Source Files

Filename	SHA256
FLYINGAVOCADOCAT.sol	97854b8b2389d6632ddfcb6e85c98aeb084404ed60aea46dd2c2d9e8b1b40d6e

Findings Breakdown



Critical	0
Medium	0
Minor / Informative	7

Severity	Unresolved	Acknowledged	Resolved	Other
Critical	0	0	0	0
Medium	0	0	0	0
Minor / Informative	7	0	0	0

Diagnostics

Critical Medium Minor / Informative

Severity	Code	Description	Status
●	IDI	Immutable Declaration Improvement	Unresolved
●	LBC	Launch Block Confusion	Unresolved
●	L02	State Variables could be Declared Constant	Unresolved
●	L04	Conformance to Solidity Naming Conventions	Unresolved
●	L09	Dead Code Elimination	Unresolved
●	L18	Multiple Pragma Directives	Unresolved
●	L19	Stable Compiler Version	Unresolved

IDI - Immutable Declaration Improvement

Criticality	Minor / Informative
Location	FLYINGAVOCADOCAT.sol#L597,602
Status	Unresolved

Description

The contract declares state variables that their value is initialized once in the constructor and are not modified afterwards. The `immutable` is a special declaration for this kind of state variables that saves gas when it is defined.

```
deployer
maxWalletBalance
```

Recommendation

By declaring a variable as immutable, the Solidity compiler is able to make certain optimizations. This can reduce the amount of storage and computation required by the contract, and make it more gas-efficient.

LBC - Launch Block Confusion

Criticality	Minor / Informative
Location	FLYINGAVOCADOCAT.sol#L620,625
Status	Unresolved

Description

The contract includes the `GROKFORLIFE` function, which allows adding liquidity to Uniswap V2. However, each time new liquidity is added, the function sets `launchBlock` to the current block number, which is misleading as the actual launch has already occurred. Additionally, after the `GROKFORLIFE` function is called, the contract prevents the sale of tokens for the next 10 blocks due to a transfer check. If the recipient is the Uniswap pair address, the `require` statement fails because the pair typically holds more tokens than the `maxWalletBalance`, effectively blocking users from selling their tokens during this period.

```
   function GROKFORLIFE() external payable {
        _approve(address(this), address(uniswapV2Router),
totalSupply());

        uniswapV2Router.addLiquidityETH{value: address(this).balance}(
            address(this),
            balanceOf(address(this)),
            0,
            0,
            deployer,
            block.timestamp
        );

        launchBlock = block.number;
    }

    function _transfer(address sender, address recipient, uint256
amount) internal override {
        if(block.number < launchBlock + 10 && sender != deployer &&
sender != address(this)){
            require(balanceOf(recipient) + amount <= maxWalletBalance,
"Exceeds max wallet balance");
        }
        super._transfer(sender, recipient, amount);
    }
```

Recommendation

Consider modifying the contract to ensure that `launchBlock` is only set once during the initial liquidity addition, preventing confusion. Additionally, review the transfer restriction logic to allow transactions involving the Uniswap pair address, ensuring users can sell tokens without being blocked for 10 blocks after `GROKFORLIFE` is called. These changes would enhance the clarity and functionality of the contract.

L02 - State Variables could be Declared Constant

Criticality	Minor / Informative
Location	FLYINGAVOCADOCAT.sol#L590
Status	Unresolved

Description

State variables can be declared as constant using the constant keyword. This means that the value of the state variable cannot be changed after it has been set. Additionally, the constant variables decrease gas consumption of the corresponding transaction.

```
uint256 public mintedSupply = 10_000_000 * 1e18
```

Recommendation

Constant state variables can be useful when the contract wants to ensure that the value of a state variable cannot be changed by any function in the contract. This can be useful for storing values that are important to the contract's behavior, such as the contract's address or the maximum number of times a certain function can be called. The team is advised to add the constant keyword to state variables that never change.

L04 - Conformance to Solidity Naming Conventions

Criticality	Minor / Informative
Location	FLYINGAVOCADOCAT.sol#L541,608
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.

```
function WETH() external pure returns (address);
function GROKFORLIFE() external payable {
...
```

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.

L09 - Dead Code Elimination

Criticality	Minor / Informative
Location	FLYINGAVOCADOCAT.sol#L421
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.

L18 - Multiple Pragma Directives

Criticality	Minor / Informative
Location	FLYINGAVOCADOCAT.sol#L10,92,122,150,516
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;
```

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	FLYINGAVOCADOCAT.sol#L10,92,122,150,516
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;
```

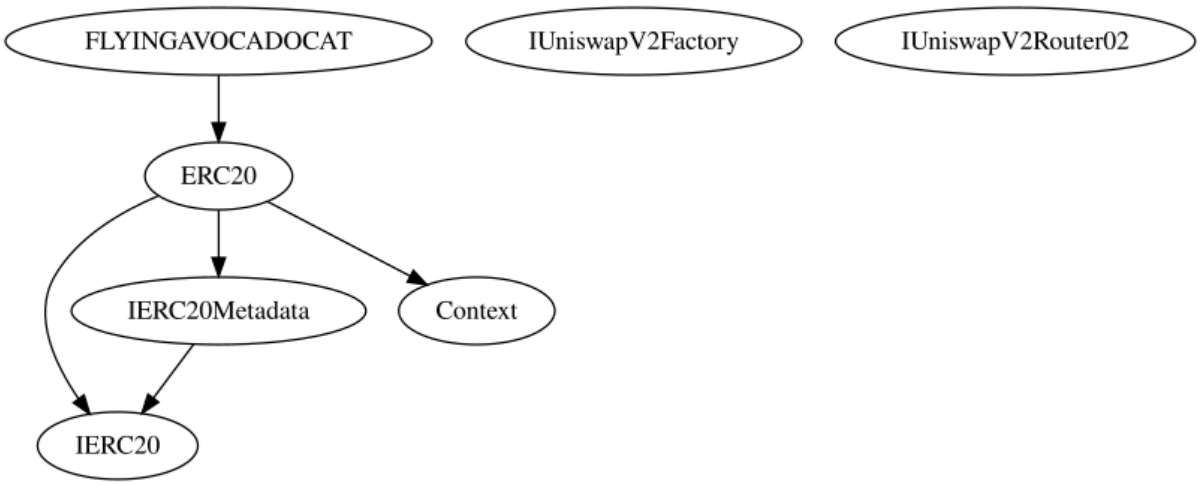
Recommendation

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.

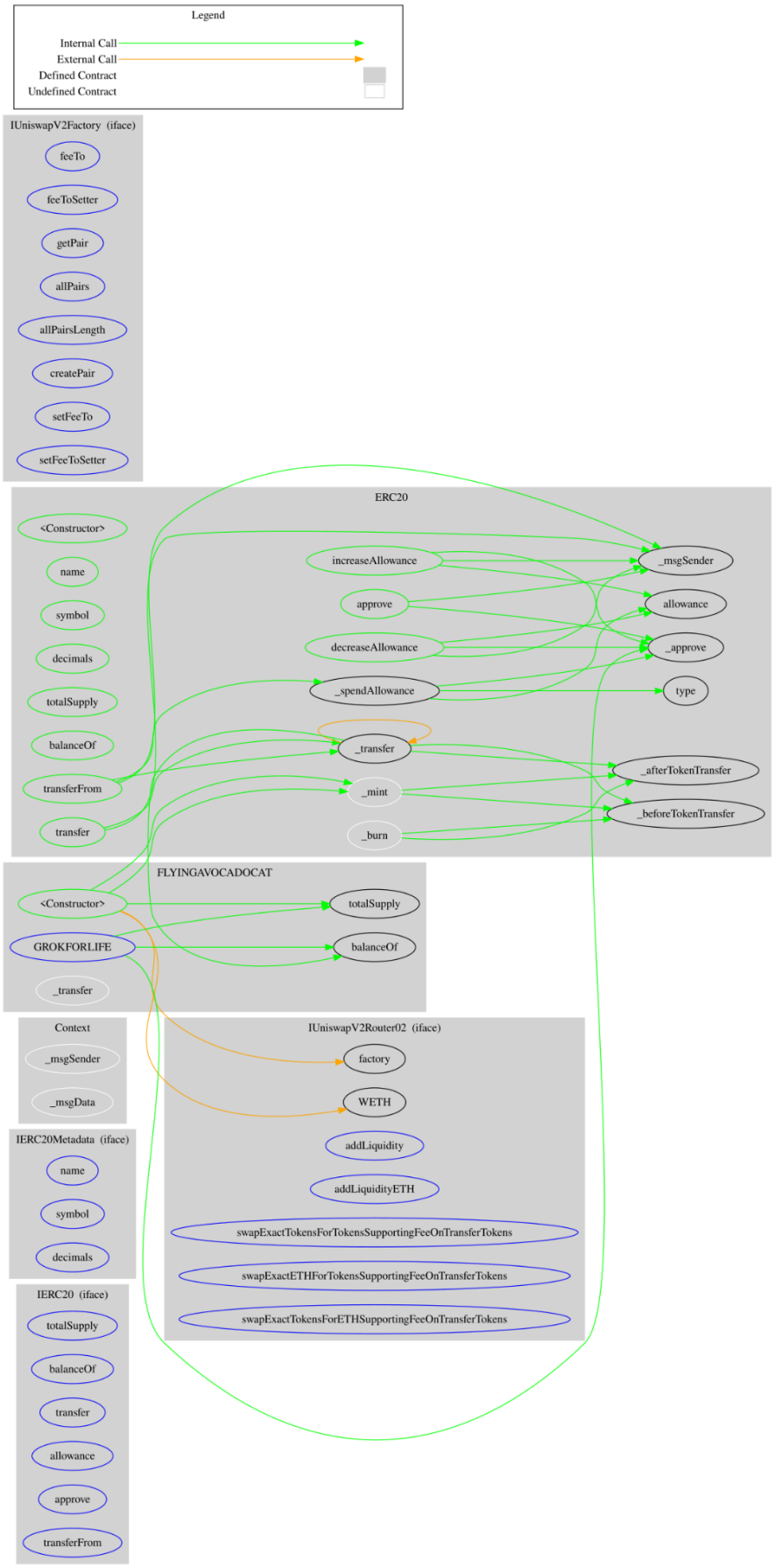
Functions Analysis

Contract	Type	Bases		
	Function Name	**Visibility**	**Mutability**	**Modifiers**
FLYINGAVOCADOCAT	Implementation	ERC20		
		Public	Payable	ERC20
	GROKFORLIFE	External	Payable	-
	_transfer	Internal	✓	

Inheritance Graph



Flow Graph


Legend
Internal Call
External Call
Defined Contract
Undefined Contract
IUniswapV2Factory (iface)
feeTo
feeToSetter
getPair
allPairs
allPairsLength
createPair
setFeeTo
setFeeToSetter
ERC20
<Constructor>
name
symbol
decimals
totalSupply
balanceOf
transferFrom
transfer
increaseAllowance
approve
decreaseAllowance
_spendAllowance
_transfer
_mint
_burn
_msgSender
allowance
_approve
type
_afterTokenTransfer
_beforeTokenTransfer
FLYINGAVOCADOCAT
<Constructor>
GROKFORLIFE
_transfer
totalSupply
balanceOf
Context
_msgSender
_msgData
IERC20Metadata (iface)
name
symbol
decimals
IERC20 (iface)
totalSupply
balanceOf
transfer
allowance
approve
transferFrom
IUniswapV2Router02 (iface)
factory
WETH
addLiquidity
addLiquidityETH
swapExactTokensForTokensSupportingFeeOnTransferTokens
swapExactETHForTokensSupportingFeeOnTransferTokens
swapExactTokensForETHSupportingFeeOnTransferTokens

Summary

Flying Avocado Cat is an interesting project that has a friendly and growing community. The Smart Contract analysis reported no compiler errors or critical issues. The contract implements a token mechanism. This audit investigates security issues, business logic concerns, and potential improvements.

Disclaimer

The information provided in this report does not constitute investment, financial or trading advice and you should not treat any of the document's content as such. This report may not be transmitted, disclosed, referred to or relied upon by any person for any purposes nor may copies be delivered to any other person other than the Company without Cyberscope's prior written consent. This report is not nor should be considered an "endorsement" or "disapproval" of any particular project or team. This report is not nor should be regarded as an indication of the economics or value of any "product" or "asset" created by any team or project that contracts Cyberscope to perform a security assessment. This document does not provide any warranty or guarantee regarding the absolute bug-free nature of the technology analyzed, nor do they provide any indication of the technologies proprietors' business, business model or legal compliance. This report should not be used in any way to make decisions around investment or involvement with any particular project. This report represents an extensive assessment process intending to help our customers increase the quality of their code while reducing the high level of risk presented by cryptographic tokens and blockchain technology.

Blockchain technology and cryptographic assets present a high level of ongoing risk Cyberscope's position is that each company and individual are responsible for their own due diligence and continuous security Cyberscope's goal is to help reduce the attack vectors and the high level of variance associated with utilizing new and consistently changing technologies and in no way claims any guarantee of security or functionality of the technology we agree to analyze. The assessment services provided by Cyberscope are subject to dependencies and are under continuing development. You agree that your access and/or use including but not limited to any services reports and materials will be at your sole risk on an as-is where-is and as-available basis Cryptographic tokens are emergent technologies and carry with them high levels of technical risk and uncertainty. The assessment reports could include false positives false negatives and other unpredictable results. The services may access and depend upon multiple layers of third parties.

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