AMAUROT

Smart Contract Audit Report

SYSFIXED

21 Oct 2022

TABLE OF CONTENTS

AUDITED DETAILS

Audited Project

Project name	Token ticker	Blockchain
AMAUROT	AMA	Binance Smart Chain

Addresses

Contract address	0xe9cd2668fb580c96b035b6d081e5753f23fe7f46
Contract deployer address	0x94fD3817270F368D563D477B917F5769eABbBd97

Project Website

https://www.meta-utopia.io/

Codebase

https://bscscan.com/address/0xe9cd2668fb580c96b035b6d081e5753f23fe7f46#code

SUMMARY

Meta-Utopia is a MetaFi project driven by community consensus to build an ideal world, an Utopia in the Metaverse. Land Cultivation and population growth paved the way for Meta Utopia, which now opens its doors to the "Era of Building and Development" in the capital city - Amaurot - the bridge for real-world businesses to enter the metaverse. The target of the Meta-Utopia is to realize a sustainable virtual ecosystem and create a peaceful environment for all its citizens.

Contract Summary

Documentation Quality

AMAUROT provides a very good documentation with standard of solidity base code.

- The technical description is provided clearly and structured and also dont have any high risk issue.

Code Quality

The Overall quality of the basecode is standard.

- Standard solidity basecode and rules are already followed by AMAUROT with the discovery of several low issues.

Test Coverage

Test coverage of the project is 100% (Through Codebase)

Audit Findings Summary

- SWC-100 SWC-108 | Explicitly define visibility for all state variables on lines 379.
- SWC-107 | It is recommended to use a reentrancy lock, reentrancy weaknesses detected on lines 620, 620, 624, 624, 515 and 690.
- SWC-110 SWC-123 | It is recommended to use of revert(), assert(), and require() in Solidity, and the new REVERT opcode in the EVM on lines 403.

CONCLUSION

We have audited the AMAUROT project released on October 2022 to discover issues and identify potential security vulnerabilities in AMAUROT Project. This process is used to find technical issues and security loopholes which might be found in the smart contract.

The security audit report provides satisfactory results with low-risk issues.

The issues found in the AMAUROT smart contract code do not pose a considerable risk. The writing of the contract is close to the standard of writing contracts in general. The low-risk issues found are some Read/write of persistent states following the external calls, state variable visibility not set, and requirement violation. The contract account state is accessed after an external call. To prevent reentrancy issues, consider accessing the state only before the call, especially if the callee is untrusted. Alternatively, a reentrancy lock can be used to prevent untrusted callees from re-entering the contract in an intermediate state. A requirement was violated in a nested call and the call was reverted as a result. Make sure valid inputs are provided to the nested call (for instance, via passed arguments).

AUDIT RESULT

Article	Category	Description	Result
Default Visibility	SWC-100 SWC-108	Functions and state variables visibility should be set explicitly. Visibility levels should be specified consciously.	ISSUE FOUND
Integer Overflow and Underflow	SWC-101	If unchecked math is used, all math operations should be safe from overflows and underflows.	PASS
Outdated Compiler Version	SWC-102	It is recommended to use a recent version of the Solidity compiler.	PASS
Floating Pragma	SWC-103	Contracts should be deployed with the same compiler version and flags that they have been tested thoroughly.	PASS
Unchecked Call Return Value	SWC-104	The return value of a message call should be checked.	PASS
Unprotected Ether Withdrawal	SWC-105	Due to missing or insufficient access controls, malicious parties can withdraw from the contract.	PASS
SELFDESTRUCT Instruction	SWC-106	The contract should not be self-destructible while it has funds belonging to users.	PASS
Reentrancy	SWC-107	Check effect interaction pattern should be followed if the code performs recursive call.	ISSUE FOUND
Uninitialized Storage Pointer	SWC-109	Uninitialized local storage variables can point to unexpected storage locations in the contract.	PASS
Assert Violation	SWC-110 SWC-123	Properly functioning code should never reach a failing assert statement.	ISSUE FOUND
Deprecated Solidity Functions	SWC-111	Deprecated built-in functions should never be used.	PASS
Delegate call to Untrusted Callee	SWC-112	Delegatecalls should only be allowed to trusted addresses.	PASS

DoS (Denial of Service)	SWC-113 SWC-128	Execution of the code should never be blocked by a specific contract state unless required.	PASS
Race Conditions	SWC-114	Race Conditions and Transactions Order Dependency should not be possible.	PASS
Authorization through tx.origin	SWC-115	tx.origin should not be used for authorization.	PASS
Block values as a proxy for time	SWC-116	Block numbers should not be used for time calculations.	PASS
Signature Unique ID	SWC-117 SWC-121 SWC-122	Signed messages should always have a unique id. A transaction hash should not be used as a unique id.	PASS
Incorrect Constructor Name	SWC-118	Constructors are special functions that are called only once during the contract creation.	PASS
Shadowing State Variable	SWC-119	State variables should not be shadowed.	PASS
Weak Sources of Randomness	SWC-120	Random values should never be generated from Chain Attributes or be predictable.	PASS
Write to Arbitrary Storage Location	SWC-124	The contract is responsible for ensuring that only authorized user or contract accounts may write to sensitive storage locations.	PASS
Incorrect Inheritance Order	SWC-125	When inheriting multiple contracts, especially if they have identical functions, a developer should carefully specify inheritance in the correct order. The rule of thumb is to inherit contracts from more /general/ to more /specific/.	PASS
Insufficient Gas Griefing	SWC-126	Insufficient gas griefing attacks can be performed on contracts which accept data and use it in a sub-call on another contract.	PASS
Arbitrary Jump Function	SWC-127	As Solidity doesnt support pointer arithmetics, it is impossible to change such variable to an arbitrary value.	PASS

Typographical Error	SWC-129	A typographical error can occur for example when the intent of a defined operation is to sum a number to a variable.	PASS
Override control character	SWC-130	Malicious actors can use the Right-To-Left-Override unicode character to force RTL text rendering and confuse users as to the real intent of a contract.	PASS
Unused variables	SWC-131 SWC-135	Unused variables are allowed in Solidity and they do not pose a direct security issue.	PASS
Unexpected Ether balance	SWC-132	Contracts can behave erroneously when they strictly assume a specific Ether balance.	PASS
Hash Collisions Variable	SWC-133	Using abi.encodePacked() with multiple variable length arguments can, in certain situations, lead to a hash collision.	PASS
Hardcoded gas amount	SWC-134	The transfer() and send() functions forward a fixed amount of 2300 gas.	PASS
Unencrypted Private Data	SWC-136	It is a common misconception that private type variables cannot be read.	PASS

SMART CONTRACT ANALYSIS

Started	Thursday Oct 20 2022 19:55:14 GMT+0000 (Coordinated Universal Time)
Finished	Friday Oct 21 2022 00:55:03 GMT+0000 (Coordinated Universal Time)
Mode	Standard
Main Source File	AMAUROT.sol

Detected Issues

ID	Title	Severity	Status
SWC-107	READ OF PERSISTENT STATE FOLLOWING EXTERNAL CALL.	low	acknowledged
SWC-107	WRITE TO PERSISTENT STATE FOLLOWING EXTERNAL CALL.	low	acknowledged
SWC-107	READ OF PERSISTENT STATE FOLLOWING EXTERNAL CALL.	low	acknowledged
SWC-107	WRITE TO PERSISTENT STATE FOLLOWING EXTERNAL CALL.	low	acknowledged
SWC-107	READ OF PERSISTENT STATE FOLLOWING EXTERNAL CALL.	low	acknowledged
SWC-107	WRITE TO PERSISTENT STATE FOLLOWING EXTERNAL CALL.	low	acknowledged
SWC-108	STATE VARIABLE VISIBILITY IS NOT SET.	low	acknowledged
SWC-123	REQUIREMENT VIOLATION.	low	acknowledged

SWC-107 | READ OF PERSISTENT STATE FOLLOWING EXTERNAL CALL.

LINE 620

low SEVERITY

The contract account state is accessed after an external call. To prevent reentrancy issues, consider accessing the state only before the call, especially if the callee is untrusted. Alternatively, a reentrancy lock can be used to prevent untrusted callees from re-entering the contract in an intermediate state.

Source File

- AMAUROT.sol

Locations

```
619
620   _balances[sender] = _balances[sender].sub(
621   amount,
622   'BEP20: transfer amount exceeds balance'
623   );
624
```

SWC-107 | WRITE TO PERSISTENT STATE FOLLOWING EXTERNAL CALL.

LINE 620

low SEVERITY

The contract account state is accessed after an external call. To prevent reentrancy issues, consider accessing the state only before the call, especially if the callee is untrusted. Alternatively, a reentrancy lock can be used to prevent untrusted callees from re-entering the contract in an intermediate state.

Source File

- AMAUROT.sol

Locations

```
619
620   _balances[sender] = _balances[sender].sub(
621   amount,
622   'BEP20: transfer amount exceeds balance'
623   );
624
```

SWC-107 | READ OF PERSISTENT STATE FOLLOWING EXTERNAL CALL.

LINE 624

low SEVERITY

The contract account state is accessed after an external call. To prevent reentrancy issues, consider accessing the state only before the call, especially if the callee is untrusted. Alternatively, a reentrancy lock can be used to prevent untrusted callees from re-entering the contract in an intermediate state.

Source File

- AMAUROT.sol

Locations

```
623   );
624   _balances[recipient] = _balances[recipient].add(amount);
625   emit Transfer(sender, recipient, amount);
626   }
627
628
```

SWC-107 | WRITE TO PERSISTENT STATE FOLLOWING EXTERNAL CALL.

LINE 624

low SEVERITY

The contract account state is accessed after an external call. To prevent reentrancy issues, consider accessing the state only before the call, especially if the callee is untrusted. Alternatively, a reentrancy lock can be used to prevent untrusted callees from re-entering the contract in an intermediate state.

Source File

- AMAUROT.sol

Locations

```
623   );
624   _balances[recipient] = _balances[recipient].add(amount);
625   emit Transfer(sender, recipient, amount);
626   }
627
628
```

SWC-107 | READ OF PERSISTENT STATE FOLLOWING EXTERNAL CALL.

LINE 515

low SEVERITY

The contract account state is accessed after an external call. To prevent reentrancy issues, consider accessing the state only before the call, especially if the callee is untrusted. Alternatively, a reentrancy lock can be used to prevent untrusted callees from re-entering the contract in an intermediate state.

Source File

- AMAUROT.sol

Locations

```
514    _msgSender(),
515    _allowances[sender][_msgSender()].sub(
516    amount,
517    'BEP20: transfer amount exceeds allowance'
518    )
519
```

SWC-107 | WRITE TO PERSISTENT STATE FOLLOWING EXTERNAL CALL.

LINE 690

low SEVERITY

The contract account state is accessed after an external call. To prevent reentrancy issues, consider accessing the state only before the call, especially if the callee is untrusted. Alternatively, a reentrancy lock can be used to prevent untrusted callees from re-entering the contract in an intermediate state.

Source File

- AMAUROT.sol

Locations

```
689
690   _allowances[owner][spender] = amount;
691   emit Approval(owner, spender, amount);
692   }
693
694
```

SWC-108 | STATE VARIABLE VISIBILITY IS NOT SET.

LINE 379

low SEVERITY

It is best practice to set the visibility of state variables explicitly. The default visibility for "_Operator" is internal. Other possible visibility settings are public and private.

Source File

- AMAUROT.sol

Locations

```
378   string public _name;
379   address _Operator;
380
381   modifier onlyOperator() {
382   require(msg.sender == _Operator, 'Permission denied');
383
```

SWC-123 | REQUIREMENT VIOLATION.

LINE 403

low SEVERITY

A requirement was violated in a nested call and the call was reverted as a result. Make sure valid inputs are provided to the nested call (for instance, via passed arguments).

Source File

- AMAUROT.sol

Locations

```
402   if (!antisnipeDisable && address(antisnipe) != address(0))
403   antisnipe.assureCanTransfer(msg.sender, from, to, amount);
404   }
405
406   function setAntisnipeDisable() external onlyOwner {
407
```

DISCLAIMER

This report is subject to the terms and conditions (including without limitation, description of services, confidentiality, disclaimer and limitation of liability) set forth in the Services Agreement, or the scope of services, and terms and conditions provided to you ("Customer" or the "Company") in connection with the Agreement. This report provided in connection with the Services set forth in the Agreement shall be used by the Company only to the extent permitted under the terms and conditions set forth in the Agreement. 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 Sysfixed's prior written consent in each instance.

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 considered, an indication of the economics or value of any "product" or "asset" created by any team or project that contracts Sysfixed to perform a security assessment. This report 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 is a limited report on our findings based on our analysis, in accordance with good industry practice as of the date of this report, in relation to cybersecurity vulnerabilities and issues in the framework and algorithms based on smart contracts, the details of which are set out in this report. In order to get a full view of our analysis, it is crucial for you to read the full report. While we have done our best in conducting our analysis and producing this report, it is important to note that you should not rely on this report and cannot claim against us on the basis of what it says or doesn't say, or how we produced it, and it is important for you to conduct your own independent investigations before making any decisions. We go into more detail on this in the below disclaimer below – please make sure to read it in full.

This report should not be used in any way to make decisions around investment or involvement with any particular project. This report in no way provides investment advice, nor should be leveraged as investment advice of any sort. This report represents an extensive assessing 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.

This report is provided for information purposes only and on a non-reliance basis and does not constitute investment advice. No one shall have any right to rely on the report or its contents, and Sysfixed and its affiliates (including holding companies, shareholders, subsidiaries, employees, directors, officers, and other representatives) (Sysfixed) owe no duty of care.

ABOUT US

Sysfixed is a blockchain security certification organization established in 2021 with the objective to provide smart contract security services and verify their correctness in blockchain-based protocols. Sysfixed automatically scans for security vulnerabilities in Ethereum and other EVM-based blockchain smart contracts. Sysfixed a comprehensive range of analysis techniques—including static analysis, dynamic analysis, and symbolic execution—can accurately detect security vulnerabilities to provide an in-depth analysis report. With a vibrant ecosystem of world-class integration partners that amplify developer productivity, Sysfixed can be utilized in all phases of your project's lifecycle. Our team of security experts is dedicated to the research and improvement of our tools and techniques used to fortify your code.