Seascape - Block Lords

Smart Contract Security Audit

Prepared by: **Halborn**

Date of Engagement: **August 15th, 2022 - August 20th, 2022**

Visit: **Halborn.com**

DOCUMENT REVISION HISTORY

VERSION	MODIFICATION	DATE	AUTHOR
0.1	Document Creation	08/15/2022	Roberto Reigada
0.2	Document Updates	08/20/2022	Roberto Reigada
0.3	Document Updates	08/20/2022	Luis Arroyo
0.4	Draft Review	08/20/2022	Gabi Urrutia
0.5	Document Updates	09/06/2022	Luis Arroyo
0.6	Draft Review	09/12/2022	Kubilay Onur Gungor
1.0	Remediation Plan	09/16/2022	Luis Arroyo
1.1	Remediation Plan Review	09/16/2022	Gabi Urrutia

CONTACTS

CONTACT	COMPANY	EMAIL
Rob Behnke	Halborn	Rob.Behnke@halborn.com
Steven Walbroehl	Halborn	Steven.Walbroehl@halborn.com

Gabi Urrutia	Halborn	Gabi.Urrutia@halborn.com
Kubilay Onur Gungor	Halborn	Kubilay.Gungor@halborn.com
Roberto Reigada	Halborn	Roberto.Reigada@halborn.com
Luis Arroyo	Halborn	Luis.Arroyo@halborn.com

EXECUTIVE OVERVIEW

1.1 INTRODUCTION

Seascape engaged Halborn to conduct a security audit on their smart contracts beginning on August 15th, 2022 and ending on August 20th, 2022. The security assessment was scoped to the smart contract provided in the GitHub repository blocklords3d/smartcontracts/

1.2 AUDIT SUMMARY

The team at Halborn was provided a week for the engagement and assigned two full-time security engineers to audit the security of the smart contract. The security engineers are blockchain and smart-contract security experts with advanced penetration testing, smart-contract hacking, and deep knowledge of multiple blockchain protocols.

The purpose of this audit is to:

- Ensure that smart contract functions operate as intended
- Identify potential security issues with the smart contracts

In summary, Halborn identified some security risks that were successfully addressed by Seascape team.

1.3 TEST APPROACH & METHODOLOGY

Halborn performed a combination of manual and automated security testing to balance efficiency, timeliness, practicality, and accuracy in regard to the scope of this audit. While manual testing is recommended to uncover flaws in logic, process, and implementation; automated testing techniques help enhance coverage of the code and can quickly identify items that do not follow the security best practices. The following phases and associated tools were used during the audit:

- Research into architecture and purpose
- Smart contract manual code review and walkthrough
- Graphing out functionality and contract logic/connectivity/functions (solgraph)
- Manual assessment of use and safety for the critical Solidity variables and functions in scope to identify any arithmetic related vulnerability classes
- Manual testing by custom scripts
- Scanning of solidity files for vulnerabilities, security hotspots or bugs. (MythX)
- Static Analysis of security for scoped contract, and imported functions. (Slither)
- Testnet deployment (Brownie, Remix IDE)

RISK METHODOLOGY:

Vulnerabilities or issues observed by Halborn are ranked based on the risk assessment methodology by measuring the **LIKELIHOOD** of a security incident and the **IMPACT** should an incident occur. This framework works for communicating the characteristics and impacts of technology vulnerabilities. The quantitative model ensures repeatable and accurate measurement while enabling users to see the underlying vulnerability characteristics that were used to generate the Risk scores. For every vulnerability, a risk level will be calculated on a scale of 5 to 1 with 5 being the highest likelihood or impact.

RISK SCALE - LIKELIHOOD

5 - Almost certain an incident will occur.
4 - High probability of an incident occurring.
3 - Potential of a security incident in the long term.
2 - Low probability of an incident occurring.
1 - Very unlikely issue will cause an incident.

RISK SCALE - IMPACT

5 - May cause devastating and unrecoverable impact or loss.
4 - May cause a significant level of impact or loss.

3 - May cause a partial impact or loss to many.
2 - May cause temporary impact or loss.
1 - May cause minimal or un-noticeable impact.

The risk level is then calculated using a sum of these two values, creating a value of 10 to 1 with 10 being the highest level of security risk.

CRITICAL	HIGH	MEDIUM	LOW	INFORMATIONAL

10 - CRITICAL
9 - 8 - HIGH
7 - 6 - MEDIUM
5 - 4 - LOW
3 - 1 - VERY LOW AND INFORMATIONAL

1.4 SCOPE

IN-SCOPE:

The security assessment was scoped to the following smart contracts

- Lord.sol
- Mead.sol

1st Commit ID: a874a71a9a07a096f82d73442e969b392056db06

2nd Commit ID: 51cf92fbaaad8d07ff4377c0a18be557ee434067

3rd Commit ID: f64fa27b972cd6697b8c851b5586b455c165aec6

4th Commit ID: 09a307a51601cfc5799b63045a22a7c1c479cc20

2. ASSESSMENT SUMMARY & FINDINGS OVERVIEW

CRITICAL	HIGH	MEDIUM	LOW	INFORMATIONAL
0	0	1	4	3

LIKELIHOOD

IMPACT

(HAL-05)	(HAL-03)	(HAL-01)		
(HAL-07)	(HAL-04)	(HAL-02)		
(HAL-08)	(HAL-06)			

SECURITY ANALYSIS	RISK LEVEL	REMEDIATION DATE
HAL01 - INTEGER OVERFLOW	Medium	SOLVED - 09/16/2022
HAL02 - EVM STACK LIMIT SURPASSED	Low	SOLVED - 09/06/2022
HAL03 - SAFEMATH LIBRARY IS NOT CORRECTLY IMPLEMENTED	Low	SOLVED - 09/16/2022
HAL04 - TOTALSUPPLY VALUE SHOULD BE OBTAINED BY TOTALSUPPLY() METHOD	Low	SOLVED - 09/06/2022
HAL05 - UNDEFINED VARIABLES ARE USED	Low	SOLVED - 09/06/2022
HAL06 - ONLYBRIDGE MODIFIER IS NEVER USED	Informational	SOLVED - 09/16/2022
HAL07 - SEEDSALE ADDRESS RECEIVES GREATER AMOUNT THAN INTENDED	Informational	SOLVED - 09/06/2022
HAL08 - FUNCTION STATE CAN BE RESTRICTED	Informational	SOLVED - 09/06/2022

FINDINGS & TECH DETAILS

3.1 (HAL-01) INTEGER OVERFLOW - MEDIUM

Description:

The Lord.sol and Mead.sol smart contracts use an insecure arithmetic operation using the totalSupply() and amount variables to determine if it is possible to mint that amount. This operation could lead to an integer overflow if the actual supply of tokens and the amount to mint are high numbers.

Code Location:

Listing 1: Lord.sol (Line 162)

```
161     function mint(address to, uint256 amount) external onlyBridge {
162         require(totalSupply() + amount <= limitSupply, "exceeded mint limit");
163         _mint(to, amount);
164     }
```

Listing 2: Mead.sol (Line 77)

```
68     function mint(uint256 _amount, uint8 _v, bytes32 _r, bytes32 _s) external {
69         // investor, project verification
70         bytes memory prefix      = "\x19Ethereum Signed Message:\n32";
71         bytes32 message          = keccak256(abi.encodePacked(msg.sender, address(this), block.chainid, _amount, mintId, mintNonceOf[msg.sender]));
72         bytes32 hash             = keccak256(abi.encodePacked(prefix, message));
73         address recover          = ecrecover(hash, _v, _r, _s);
74
75         require(bridges[recover], "sig");
76
```

```
        require(totalSupply() + _amount <= limitSupply, "exceeded mint limit");

        mintNonceOf[msg.sender]++;

        _mint(msg.sender, _amount);
    }
```

Proof of Concept:

to replicate this issue:

- in lord.sol:
  - increase limit supply by any number.
  - try to mint an amount which could cause an overflow, for example ‘0xfffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffa’.
- in mead.sol:
  - mint a high amount, for example ‘0xfffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffa’.
  - mint again any amount greater than 5 to cause overflow.

Listing 3: pentest.js

```
    let amount1 = '0xfffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffa';
    let amount2 = '7';

    //..snipped..

    await mead.connect(bridge).mint(amount1, sig.v, sig.r, sig.s);
    await mead.connect(bridge).mint(amount2, sig.v, sig.r, sig.s);
```

Listing 4: Output

```
Error: VM Exception while processing transaction: reverted with panic code 0x11 (Arithmetic operation underflowed or overflowed
↳ outside an unchecked block)
    at Mead.mint (contracts/erc20/Mead.sol:78)
```

Risk Level:

Likelihood - 3
Impact - 3

Recommendation:

It is recommended to import the OpenZeppelin SafeMath.sol library and set the variables using that SafeMath to avoid these extreme situations. This could be done by adding the following lines to the contracts.

Listing 5: safemath.sol

```
import "@openzeppelin/contracts/utils/math/SafeMath.sol";
using SafeMath for uint256;
```

Remediation Plan:

SOLVED: The SeaScape Team now implements correctly the SafeMath library to avoid these overflows.

3.2 (HAL-02) EVM STACK LIMIT SURPASSED - LOW

Description:

The EVM stack is a maximum of 16 deep. Every variable that is created will get pushed onto the stack. This includes function parameters and local variables. The Lord constructor uses too many parameters and local variables, which causes the following error to be displayed at compile time.

Listing 6: StackLimit

```
CompilerError: Stack too deep when compiling inline assembly:
↳ Variable headStart is 1 slot(s) too deep inside the stack.
```

Code Location:

Listing 7: Lord.sol

```
constructor(
        address _seedSale,
        address _strategicSale,
        address _privateSale,
        address _launchpads,
        address _ieo,
        address _lordsBounty,
        address _kingsBounty,
        address _dynastyIncentives,
        address _liquidity,
        address _foundationReserve,
        address _advisor,
        bool _bridgeAllowed) ERC20("BLOCKLORDS", "LORD") {
        bridgeAllowed = _bridgeAllowed;
        uint256 _million = 1000 * 1000 * 10 ** 18;
        uint256 thousand = 1000 * 10 ** 18;

        if (!_bridgeAllowed) {
            _mint(_seedSale, 8 * _million + (750 * thousand));  // 8.75% of 100 million
            _mint(_seedSale, 6 * _million + (250 * thousand));  // 8.75% of 100 million
            _mint(_privateSale, 7 * _million);  // 8.75% of 100 million
            _mint(_launchpads, 2 * _million);  // 8.75% of 100 million
            _mint(_ieo, 1 * _million);  // 8.75% of 100 million
            _mint(_lordsBounty, 25 * _million);  // 8.75% of 100 million
            _mint(_kingsBounty, 10 * _million);  // 8.75% of 100 million
            _mint(_dynastyIncentives, 15 * _million);  // 8.75% of 100 million
            _mint(_liquidity, 10 * _million);  // 8.75% of 100 million
            _mint(_foundationReserve, 10 * _million);  // 8.75% of 100 million
            _mint(_advisor, 5 * _million);  // 8.75% of 100 million

            require(totalSupply() == 100 * _million, "not a 100 million tokens");
        }
    }
```

Risk Level:

Likelihood - 3

Impact - 2

Recommendation:

It is recommended to refactor the parameters of the smart contract constructor. The use of structures containing variables that can be bundled together is recommended. It could also be feasible to execute part of the instructions in a new function called inside the constructor.

Remediation Plan:

SOLVED: The minting process is now done address by address in several new functions.

3.3 (HAL-03) SAFEMATH LIBRARY IS NOT CORRECTLY IMPLEMENTED - LOW

Description:

The Lord.sol and Mead.sol smart contracts use .add() and .sub() functions located in the OpenZeppelin SafeMath library. This library is neither imported nor associated to a variable type (in this case uint256), so the mentioned functions cannot be used.

Code Location:

Listing 8: Lord.sol (Line 93)

```
92 function mint(address to, uint256 amount) external onlyBridge {
93         require(totalSupply.add(amount) <= limitSupply, "exceeded mint limit");
94         _mint(to, amount);
95     }
```

Listing 9: Lord.sol (Lines 123,124)

```
119 function burnFrom(address account, uint256 amount) public onlyBridge {
120         uint256 currentAllowance = allowance(account, _msgSender());
121         require(currentAllowance >= amount, "burn amount exceeds allowance");
122
123         _approve(account, _msgSender(), currentAllowance
124             .sub(amount, "transfer amount exceeds allowance"));
125         _burn(account, amount);
126     }
```

Listing 10: Mead.sol (Line 73)

```
63     function mint(uint256 _amount, uint8 _v, bytes32 _r, bytes32 _s) external {
```

```solidity
        // investor, project verification
        bytes memory prefix      = "\x19Ethereum Signed Message:\n32";
        bytes32 message          = keccak256(abi.encodePacked(msg.sender, address(this), chainid, _amount, mintId, mintNonceOf[msg.sender]));
        bytes32 hash            = keccak256(abi.encodePacked(prefix, message));
        address recover          = ecrecover(hash, _v, _r, _s);

        require(bridges[recover], "sig");

        require(_totalSupply.add(amount) <= limitSupply, "exceeded mint limit");

        mintNonceOf[msg.sender]++;

        _mint(msg.sender, _amount);
    }
```

Risk Level:

Likelihood - 2
Impact - 3

Recommendation:

It is recommended to import the OpenZeppelin SafeMath.sol library and set the variables that are using that SafeMath. This could be done by adding the following lines to the contracts.

Listing 11: safemath.sol

```solidity
import "@openzeppelin/contracts/utils/math/SafeMath.sol";
using SafeMath for uint256;
```

Remediation Plan:

SOLVED: The SeaScape Team now implements correctly the SafeMath library.

3.4 (HAL-04) TOTALSUPPLY VALUE SHOULD BE OBTAINED BY TOTALSUPPLY() METHOD - LOW

Description:

The Lord.sol and Mead.sol smart contracts use _totalSupply variable to obtain the tokens total supply. This value is defined as a private variable in the OpenZeppelin ERC20 implementation; therefore, it should be obtained by using the get method
totalSupply().

Code Location:

Listing 12: Lord.sol

```
require(totalSupply == 100 * _million, "not a 100 million tokens")
↳ ;
```

Listing 13: Lord.sol

```
require(totalSupply.add(amount) <= limitSupply, "exceeded mint
↳ limit");
```

Listing 14: Mead.sol

```
require(_totalSupply.add(amount) <= limitSupply, "exceeded mint
↳ limit");
```

Risk Level:

Likelihood - 2
Impact - 2

Recommendation:

It is recommended to use the get method totalSupply() to retrieve the token total supply.

Remediation Plan:

SOLVED: The SeaScape team now uses the totalSupply() method to retrieve the total token supply.

3.5 (HAL-05) UNDEFINED VARIABLES ARE USED - LOW

Description:

The Lord.sol, Mead.sol and ImportExportElasticNft.sol smart contracts use undefined variables, resulting in contracts which do not compile.

Code Location:

- limitSupply (Lord.sol#93)
- bridgeAllowed (Mead.sol#36,46)
- amount (Mead.sol#73)
- memory_amount (Mead.sol#98)
- chainid (Mead.sol#67,90)

Risk Level:

Likelihood - 1
Impact - 3

Recommendation:

It is recommended to declare all used variables. In the case of the chain ID variable, it is recommended to recalculate it each time it is used because its value could change in case of a fork. For this purpose, block.chainid could be used instead of creating a variable.

Remediation Plan:

SOLVED: The SeaScape Team now implements the mentioned variables.

3.6 (HAL-06) ONLYBRIDGE MODIFIER IS NEVER USED - INFORMATIONAL

Description:

The onlyBridge modifier is never used in the code.

Code Location:

Listing 15: Mead.sol

```
    modifier onlyBridge {
            require(bridges[msg.sender]);
        _;
    }
```

Risk Level:

Likelihood - 2
Impact - 1

Recommendation:

It is recommended to remove or comment the unused code from the contracts.

Remediation Plan:

SOLVED: The SeaScape Team now uses the onlyBridge modifier on the mint and burn functions.

3.7 (HAL-07) SEEDSALE ADDRESS RECEIVES GREATER AMOUNT THAN INTENDED - INFORMATIONAL

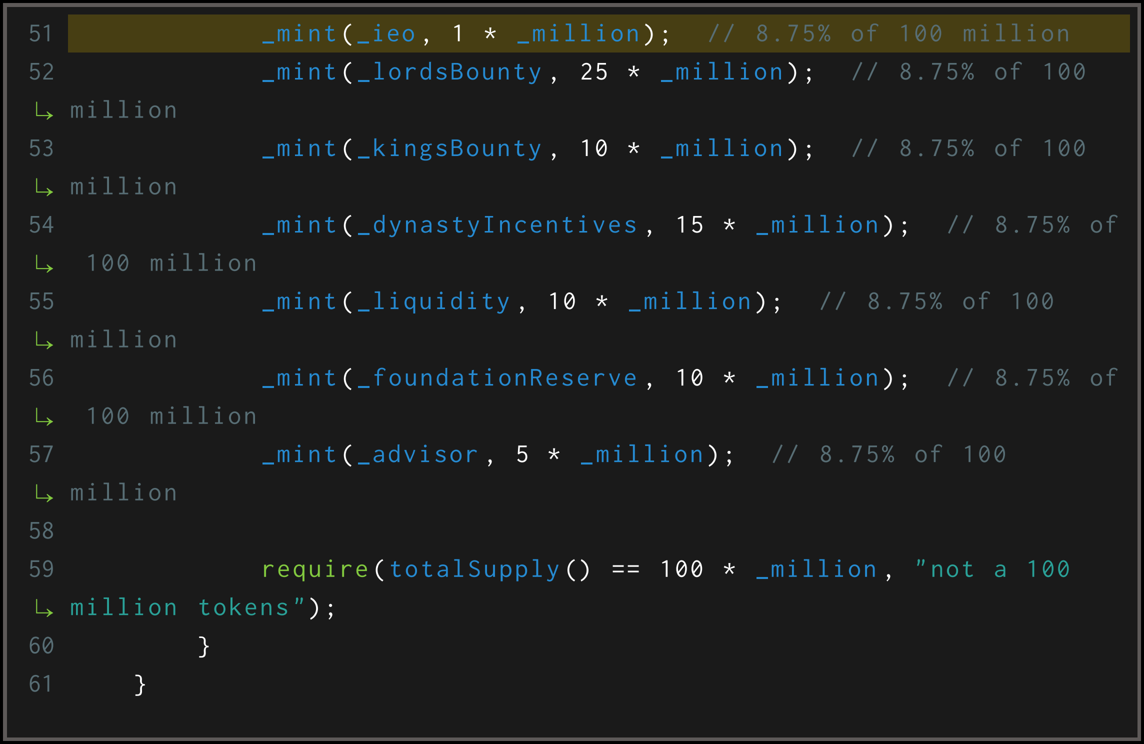
Description:

Several amounts are minted to the accounts, added as arguments in the constructor. Different amounts are minted twice in the _seedsale account, making this account 15 million instead of 8.75 million.

Code Location:

Listing 16: Lord.sol (Lines 50,51)

```
29 constructor(
30         address _seedSale,
31         address _strategicSale,
32         address _privateSale,
33         address _launchpads,
34         address _ieo,
35         address _lordsBounty,
36         address _kingsBounty,
37         address _dynastyIncentives,
38         address _liquidity,
39         address _foundationReserve,
40         address _advisor,
41         bool _bridgeAllowed) ERC20("BLOCKLORDS", "LORD") {
42         bridgeAllowed = _bridgeAllowed;
43         uint256 _million = 1000 * 1000 * 10 ** 18;
44         uint256 thousand = 1000 * 10 ** 18;
45
46         if (!_bridgeAllowed) {
47             _mint(_seedSale, 8 * _million + (750 * thousand));  //
   ↳  8.75% of 100 million
48             _mint(_seedSale, 6 * _million + (250 * thousand));  //
   ↳  8.75% of 100 million
49             _mint(_privateSale, 7 * _million);  // 8.75% of 100
   ↳ million
50             _mint(_launchpads, 2 * _million);  // 8.75% of 100
   ↳ million
```



```
            _mint(_ieo, 1 * _million);  // 8.75% of 100 million
            _mint(_lordsBounty, 25 * _million);  // 8.75% of 100 million
            _mint(_kingsBounty, 10 * _million);  // 8.75% of 100 million
            _mint(_dynastyIncentives, 15 * _million);  // 8.75% of 100 million
            _mint(_liquidity, 10 * _million);  // 8.75% of 100 million
            _mint(_foundationReserve, 10 * _million);  // 8.75% of 100 million
            _mint(_advisor, 5 * _million);  // 8.75% of 100 million

            require(totalSupply() == 100 * _million, "not a 100 million tokens");
        }
    }
```

Risk Level:

Likelihood - 1
Impact - 2

Recommendation:

It is recommended that you review the amounts that are minted to each account.

Remediation Plan:

SOLVED: The seedsale address now receives the correct amount of tokens.

3.8 (HAL-08) FUNCTION STATE CAN BE RESTRICTED - INFORMATIONAL

Description:

The state mutability of the burn() function can be restricted to pure.

Code Location:

Listing 17: Lord.sol

```
    function burn(uint256 amount) public {
        require(false, "Only burnFrom is allowed");
    }
```

Risk Level:

Likelihood - 1
Impact - 1

Recommendation:

It is recommended to restrict the state of the function to pure for saving gas.

Remediation Plan:

SOLVED: The SeaScape team has removed the affected function.

AUTOMATED TESTING

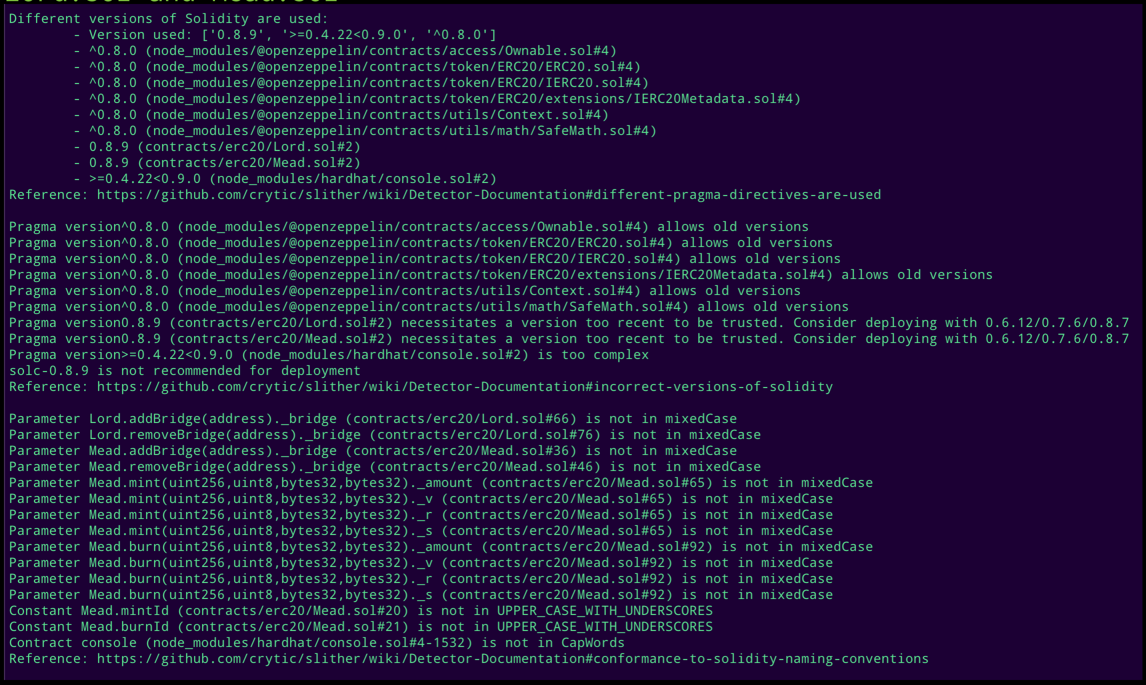
4.1 STATIC ANALYSIS REPORT

Description:

Halborn used automated testing techniques to enhance the coverage of certain areas of the smart contracts in scope. Among the tools used was Slither, a Solidity static analysis framework. After Halborn verified the smart contracts in the repository and was able to compile them correctly into their ABIs and binary format, Slither was run against the contracts. This tool can statically verify mathematical relationships between Solidity variables to detect invalid or inconsistent usage of the contracts’ APIs across the entire code-base.

Slither results:

Lord.sol and Mead.sol

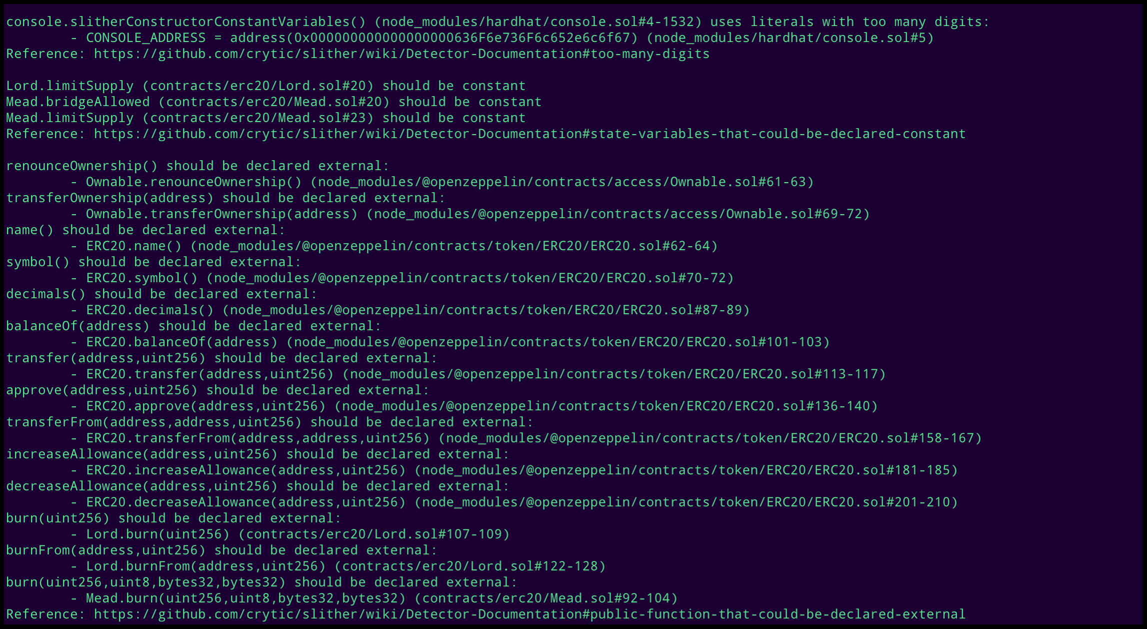


```
Different versions of Solidity are used:
        - Version used: ['0.8.9', '>=0.4.22<0.9.0', '^0.8.0']
        - ^0.8.0 (node_modules/@openzeppelin/contracts/access/Ownable.sol#4)
        - ^0.8.0 (node_modules/@openzeppelin/contracts/token/ERC20/ERC20.sol#4)
        - ^0.8.0 (node_modules/@openzeppelin/contracts/token/ERC20/IERC20.sol#4)
        - ^0.8.0 (node_modules/@openzeppelin/contracts/token/ERC20/extensions/IERC20Metadata.sol#4)
        - ^0.8.0 (node_modules/@openzeppelin/contracts/utils/Context.sol#4)
        - ^0.8.0 (node_modules/@openzeppelin/contracts/utils/math/SafeMath.sol#4)
        - 0.8.9 (contracts/erc20/Lord.sol#2)
        - 0.8.9 (contracts/erc20/Mead.sol#2)
        - >=0.4.22<0.9.0 (node_modules/hardhat/console.sol#2)
Reference: https://github.com/crytic/slither/wiki/Detector-Documentation#different-pragma-directives-are-used

Pragma version^0.8.0 (node_modules/@openzeppelin/contracts/access/Ownable.sol#4) allows old versions
Pragma version^0.8.0 (node_modules/@openzeppelin/contracts/token/ERC20/ERC20.sol#4) allows old versions
Pragma version^0.8.0 (node_modules/@openzeppelin/contracts/token/ERC20/IERC20.sol#4) allows old versions
Pragma version^0.8.0 (node_modules/@openzeppelin/contracts/token/ERC20/extensions/IERC20Metadata.sol#4) allows old versions
Pragma version^0.8.0 (node_modules/@openzeppelin/contracts/utils/Context.sol#4) allows old versions
Pragma version^0.8.0 (node_modules/@openzeppelin/contracts/utils/math/SafeMath.sol#4) allows old versions
Pragma version0.8.9 (contracts/erc20/Lord.sol#2) necessitates a version too recent to be trusted. Consider deploying with 0.6.12/0.7.6/0.8.7
Pragma version0.8.9 (contracts/erc20/Mead.sol#2) necessitates a version too recent to be trusted. Consider deploying with 0.6.12/0.7.6/0.8.7
Pragma version>=0.4.22<0.9.0 (node_modules/hardhat/console.sol#2) is too complex
solc-0.8.9 is not recommended for deployment
Reference: https://github.com/crytic/slither/wiki/Detector-Documentation#incorrect-versions-of-solidity

Parameter Lord.addBridge(address)._bridge (contracts/erc20/Lord.sol#66) is not in mixedCase
Parameter Lord.removeBridge(address)._bridge (contracts/erc20/Lord.sol#76) is not in mixedCase
Parameter Mead.addBridge(address)._bridge (contracts/erc20/Mead.sol#36) is not in mixedCase
Parameter Mead.removeBridge(address)._bridge (contracts/erc20/Mead.sol#46) is not in mixedCase
Parameter Mead.mint(uint256,uint8,bytes32,bytes32)._amount (contracts/erc20/Mead.sol#65) is not in mixedCase
Parameter Mead.mint(uint256,uint8,bytes32,bytes32)._v (contracts/erc20/Mead.sol#65) is not in mixedCase
Parameter Mead.mint(uint256,uint8,bytes32,bytes32)._r (contracts/erc20/Mead.sol#65) is not in mixedCase
Parameter Mead.mint(uint256,uint8,bytes32,bytes32)._s (contracts/erc20/Mead.sol#65) is not in mixedCase
Parameter Mead.burn(uint256,uint8,bytes32,bytes32)._amount (contracts/erc20/Mead.sol#92) is not in mixedCase
Parameter Mead.burn(uint256,uint8,bytes32,bytes32)._v (contracts/erc20/Mead.sol#92) is not in mixedCase
Parameter Mead.burn(uint256,uint8,bytes32,bytes32)._r (contracts/erc20/Mead.sol#92) is not in mixedCase
Parameter Mead.burn(uint256,uint8,bytes32,bytes32)._s (contracts/erc20/Mead.sol#92) is not in mixedCase
Constant Mead.mintId (contracts/erc20/Mead.sol#20) is not in UPPER_CASE_WITH_UNDERSCORES
Constant Mead.burnId (contracts/erc20/Mead.sol#21) is not in UPPER_CASE_WITH_UNDERSCORES
Contract console (node_modules/hardhat/console.sol#4-1532) is not in CapWords
Reference: https://github.com/crytic/slither/wiki/Detector-Documentation#conformance-to-solidity-naming-conventions
```

AUTOMATED TESTING



```
console.slitherConstructorConstantVariables() (node_modules/hardhat/console.sol#4-1532) uses literals with too many digits:
	- CONSOLE_ADDRESS = address(0x000000000000000000636F6e736F6c652e6c6f67) (node_modules/hardhat/console.sol#5)
Reference: https://github.com/crytic/slither/wiki/Detector-Documentation#too-many-digits

Lord.limitSupply (contracts/erc20/Lord.sol#20) should be constant
Mead.bridgeAllowed (contracts/erc20/Mead.sol#20) should be constant
Mead.limitSupply (contracts/erc20/Mead.sol#23) should be constant
Reference: https://github.com/crytic/slither/wiki/Detector-Documentation#state-variables-that-could-be-declared-constant

renounceOwnership() should be declared external:
	- Ownable.renounceOwnership() (node_modules/@openzeppelin/contracts/access/Ownable.sol#61-63)
transferOwnership(address) should be declared external:
	- Ownable.transferOwnership(address) (node_modules/@openzeppelin/contracts/access/Ownable.sol#69-72)
name() should be declared external:
	- ERC20.name() (node_modules/@openzeppelin/contracts/token/ERC20/ERC20.sol#62-64)
symbol() should be declared external:
	- ERC20.symbol() (node_modules/@openzeppelin/contracts/token/ERC20/ERC20.sol#70-72)
decimals() should be declared external:
	- ERC20.decimals() (node_modules/@openzeppelin/contracts/token/ERC20/ERC20.sol#87-89)
balanceOf(address) should be declared external:
	- ERC20.balanceOf(address) (node_modules/@openzeppelin/contracts/token/ERC20/ERC20.sol#101-103)
transfer(address,uint256) should be declared external:
	- ERC20.transfer(address,uint256) (node_modules/@openzeppelin/contracts/token/ERC20/ERC20.sol#113-117)
approve(address,uint256) should be declared external:
	- ERC20.approve(address,uint256) (node_modules/@openzeppelin/contracts/token/ERC20/ERC20.sol#136-140)
transferFrom(address,address,uint256) should be declared external:
	- ERC20.transferFrom(address,address,uint256) (node_modules/@openzeppelin/contracts/token/ERC20/ERC20.sol#158-167)
increaseAllowance(address,uint256) should be declared external:
	- ERC20.increaseAllowance(address,uint256) (node_modules/@openzeppelin/contracts/token/ERC20/ERC20.sol#181-185)
decreaseAllowance(address,uint256) should be declared external:
	- ERC20.decreaseAllowance(address,uint256) (node_modules/@openzeppelin/contracts/token/ERC20/ERC20.sol#201-210)
burn(uint256) should be declared external:
	- Lord.burn(uint256) (contracts/erc20/Lord.sol#107-109)
burnFrom(address,uint256) should be declared external:
	- Lord.burnFrom(address,uint256) (contracts/erc20/Lord.sol#122-128)
burn(uint256,uint8,bytes32,bytes32) should be declared external:
	- Mead.burn(uint256,uint8,bytes32,bytes32) (contracts/erc20/Mead.sol#92-104)
Reference: https://github.com/crytic/slither/wiki/Detector-Documentation#public-function-that-could-be-declared-external
```

- No major issues found by Slither.

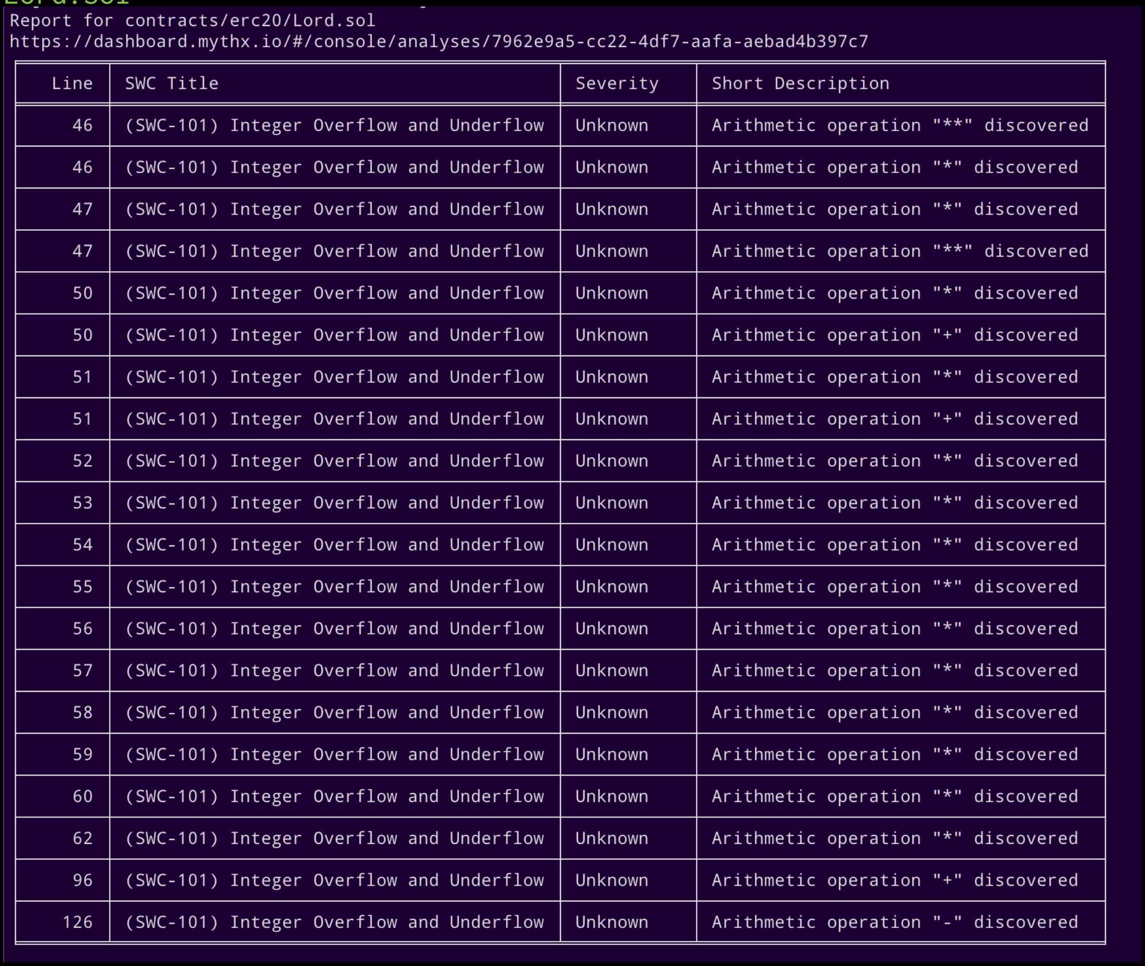
4.2 AUTOMATED SECURITY SCAN

Description:

Halborn used automated security scanners to assist with detection of well-known security issues and to identify low-hanging fruits on the targets for this engagement. Among the tools used was MythX, a security analysis service for Ethereum smart contracts. MythX performed a scan on the smart contracts and sent the compiled results to the analyzers to locate any vulnerabilities.

MythX results:

Lord.sol

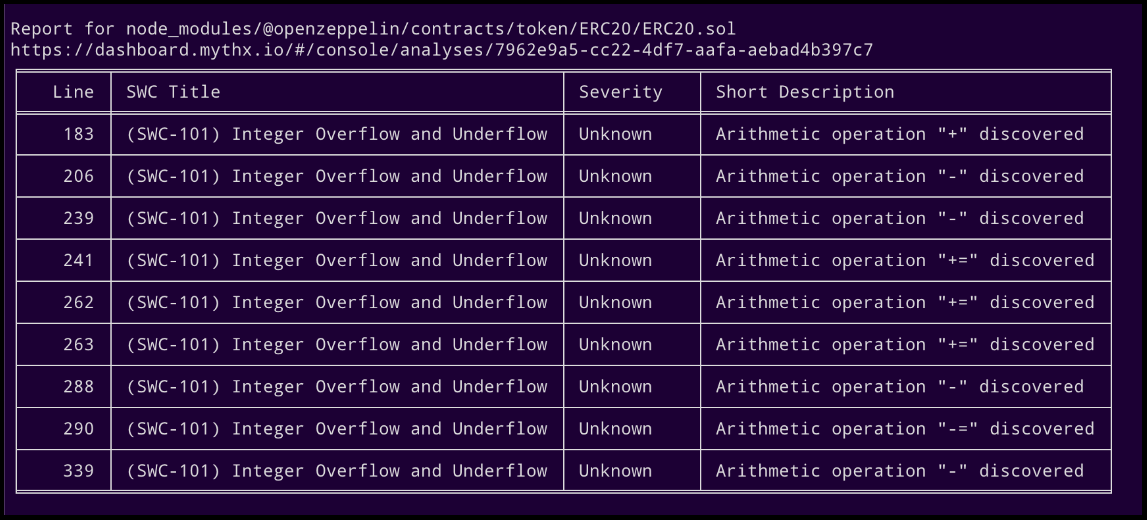


```
Report for contracts/erc20/Lord.sol
https://dashboard.mythx.io/#/console/analyses/7962e9a5-cc22-4df7-aafa-aebad4b397c7

  Line   SWC Title                                  Severity   Short Description
    46   (SWC-101) Integer Overflow and Underflow   Unknown    Arithmetic operation "**" discovered
    46   (SWC-101) Integer Overflow and Underflow   Unknown    Arithmetic operation "*" discovered
    47   (SWC-101) Integer Overflow and Underflow   Unknown    Arithmetic operation "*" discovered
    47   (SWC-101) Integer Overflow and Underflow   Unknown    Arithmetic operation "**" discovered
    50   (SWC-101) Integer Overflow and Underflow   Unknown    Arithmetic operation "*" discovered
    50   (SWC-101) Integer Overflow and Underflow   Unknown    Arithmetic operation "+" discovered
    51   (SWC-101) Integer Overflow and Underflow   Unknown    Arithmetic operation "*" discovered
    51   (SWC-101) Integer Overflow and Underflow   Unknown    Arithmetic operation "+" discovered
    52   (SWC-101) Integer Overflow and Underflow   Unknown    Arithmetic operation "*" discovered
    53   (SWC-101) Integer Overflow and Underflow   Unknown    Arithmetic operation "*" discovered
    54   (SWC-101) Integer Overflow and Underflow   Unknown    Arithmetic operation "*" discovered
    55   (SWC-101) Integer Overflow and Underflow   Unknown    Arithmetic operation "*" discovered
    56   (SWC-101) Integer Overflow and Underflow   Unknown    Arithmetic operation "*" discovered
    57   (SWC-101) Integer Overflow and Underflow   Unknown    Arithmetic operation "*" discovered
    58   (SWC-101) Integer Overflow and Underflow   Unknown    Arithmetic operation "*" discovered
    59   (SWC-101) Integer Overflow and Underflow   Unknown    Arithmetic operation "*" discovered
    60   (SWC-101) Integer Overflow and Underflow   Unknown    Arithmetic operation "*" discovered
    62   (SWC-101) Integer Overflow and Underflow   Unknown    Arithmetic operation "*" discovered
    96   (SWC-101) Integer Overflow and Underflow   Unknown    Arithmetic operation "+" discovered
   126   (SWC-101) Integer Overflow and Underflow   Unknown    Arithmetic operation "-" discovered
```

Mead.sol

AUTOMATED TESTING



```
Report for node_modules/@openzeppelin/contracts/token/ERC20/ERC20.sol
https://dashboard.mythx.io/#/console/analyses/7962e9a5-cc22-4df7-aafa-aebad4b397c7
╒════════╤════════════════════════════════════════╤════════════╤═══════════════════════════════════╕
│   Line │ SWC Title                              │ Severity   │ Short Description                 │
╞════════╪════════════════════════════════════════╪════════════╪═══════════════════════════════════╡
│    183 │ (SWC-101) Integer Overflow and Underflow │ Unknown  │ Arithmetic operation "+" discovered │
├────────┼────────────────────────────────────────┼────────────┼───────────────────────────────────┤
│    206 │ (SWC-101) Integer Overflow and Underflow │ Unknown  │ Arithmetic operation "-" discovered │
├────────┼────────────────────────────────────────┼────────────┼───────────────────────────────────┤
│    239 │ (SWC-101) Integer Overflow and Underflow │ Unknown  │ Arithmetic operation "-" discovered │
├────────┼────────────────────────────────────────┼────────────┼───────────────────────────────────┤
│    241 │ (SWC-101) Integer Overflow and Underflow │ Unknown  │ Arithmetic operation "+=" discovered │
├────────┼────────────────────────────────────────┼────────────┼───────────────────────────────────┤
│    262 │ (SWC-101) Integer Overflow and Underflow │ Unknown  │ Arithmetic operation "+=" discovered │
├────────┼────────────────────────────────────────┼────────────┼───────────────────────────────────┤
│    263 │ (SWC-101) Integer Overflow and Underflow │ Unknown  │ Arithmetic operation "+=" discovered │
├────────┼────────────────────────────────────────┼────────────┼───────────────────────────────────┤
│    288 │ (SWC-101) Integer Overflow and Underflow │ Unknown  │ Arithmetic operation "-" discovered │
├────────┼────────────────────────────────────────┼────────────┼───────────────────────────────────┤
│    290 │ (SWC-101) Integer Overflow and Underflow │ Unknown  │ Arithmetic operation "-=" discovered │
├────────┼────────────────────────────────────────┼────────────┼───────────────────────────────────┤
│    339 │ (SWC-101) Integer Overflow and Underflow │ Unknown  │ Arithmetic operation "-" discovered │
╘════════╧════════════════════════════════════════╧════════════╧═══════════════════════════════════╛
```

- No major issues found by MythX.

THANK YOU FOR CHOOSING

HALBORN