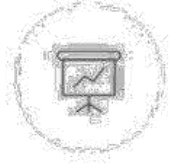




ITISH
AUDIT COMPANY

Audit Details



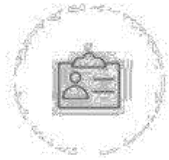
Contract Name

Acossi Coin



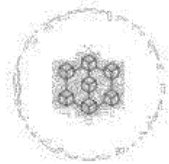
Deployer address

0x4aaefa7cfd5f5d6e630f29c76a3e17ef02723195



Client contacts:

Acossi Coin team



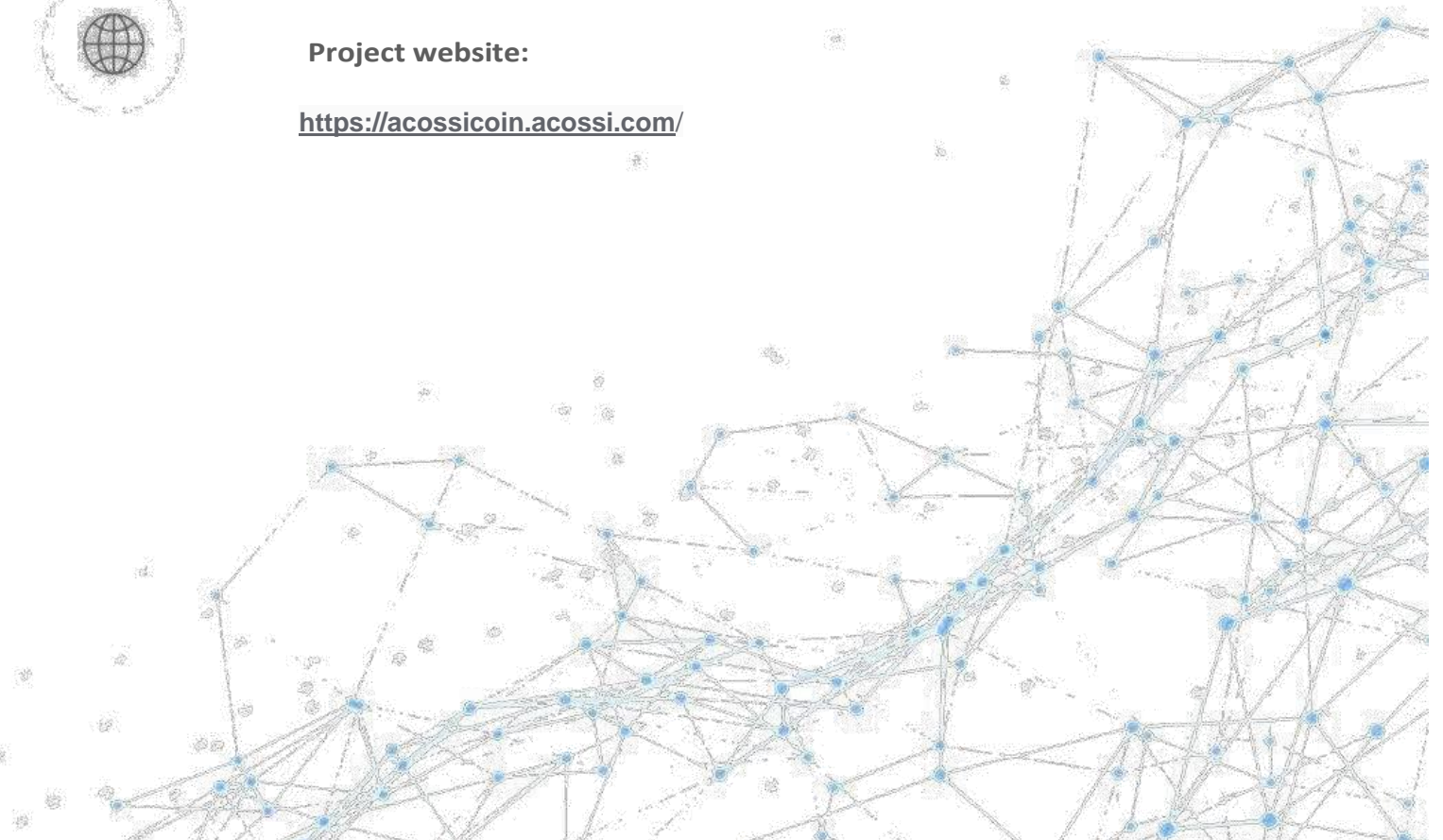
Blockchain

Binance



Project website:

<https://acossicoin.acossi.com/>



Disclaimer

This is a limited report on our findings based on our analysis, in accordance with good industry practice as at 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.

DISCLAIMER: By reading this report or any part of it, you agree to the terms of this disclaimer. If you do not agree to the terms, then please immediately cease reading this report, and delete and destroy any and all copies of this report downloaded and/or printed by you. 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 Itish and its affiliates (including holding companies, shareholders, subsidiaries, employees, directors, officers and other representatives) (Itish) owe no duty of care towards you or any other person, nor does Itish make any warranty or representation to any person on the accuracy or completeness of the report. The report is provided "as is", without any conditions, warranties or other terms of any kind except as set out in this disclaimer, and Itish hereby excludes all representations, warranties, conditions and other terms (including, without limitation, the warranties implied by law of satisfactory quality, fitness for purpose and the use of reasonable care and skill) which, but for this clause, might have effect in relation to the report. Except and only to the extent that it is prohibited by law, Itish hereby excludes all liability and responsibility, and neither you nor any other person shall have any claim against Itish, for any amount or kind of loss or damage that may result to you or any other person (including without limitation, any direct, indirect, special, punitive, consequential or pure economic loss or damages, or any loss of income, profits, goodwill, data, contracts, use of money, or business interruption, and whether in delict, tort (including without limitation negligence), contract, breach of statutory duty, misrepresentation (whether innocent or negligent) or otherwise under any claim of any nature whatsoever in any jurisdiction) in any way arising from or connected with this report and the use, inability to use or the results of use of this report, and any reliance on this report.

The analysis of the security is purely based on the smart contracts alone. No applications or operations were reviewed for security. No product code has been reviewed.

Background

Itish was commissioned Acossi Coin token to perform an audit of smart contracts:

<https://bscscan.com/token/0x4aaefa7cfd5f5d6e630f29c76a3e17ef02723195>

The purpose of the audit was to achieve the following:

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














The information in this report should be used to understand the risk exposure of the smart contract, and as a guide to improve the security posture of the smart contract by remediating the issues that were identified.

Contract Details

Token contract details for 07.09.2024

contract name	Acossi Coin
Contract creator	0x4aaefa7cfd5f5d6e630f29c76a3e17ef02723195
Transaction's count	2
Decimals	18

Contract Top Transactions

 Transaction Hash	Method 	Block	Age	From	To	Amount
 0xb881826e85... 	Deposit	41056975	34 days ago	0xD156217D...5D20e16Cb 	 0xA4a4F6b7...2aFb17bb0 	400,000,000
 0xc40ad59278... 	0x60806040	40748448	44 days ago	 Null: 0x000...000 	 0xD156217D...5D20e16Cb 	1,000,000,000

Token Functions Details

```
msgSender()  
msgData()  
totalSupply()  
getOwner()  
balanceOf()  
transfer()  
allowance()  
approval()  
transferFrom()  
name()  
symbol()
```

Contract Interface Details

```
interface IERC20  
interface IERC20 Metadata is IERC20
```

Issues Checking Status

Issue description	checking status
1. Compiler errors.	Passed
2. Compiler Compatibilities	Passe
3. Possible delays in data delivery.	Passed
4. Oracle calls.	Moderate
5. Front running.	Failed
6. Timestamp dependence.	Passed
7. Integer Overflow and Underflow.	Failed
8. DoS with Revert.	Severe
9. DoS with block gas limit.	Moderate
10. Methods execution permissions.	Passed
11. Economy model of the contract.	Passed
12. The impact of the exchange rate on the logic.	Severe
13. Private user data leaks.	Passed
14. Malicious Event log.	Passed
15. Scoping and Declarations.	Passed
16. Uninitialized storage pointers.	Passed
17. Arithmetic accuracy.	Moderate
18. Design Logic.	poor

19. Cross-function race conditions.

Passed

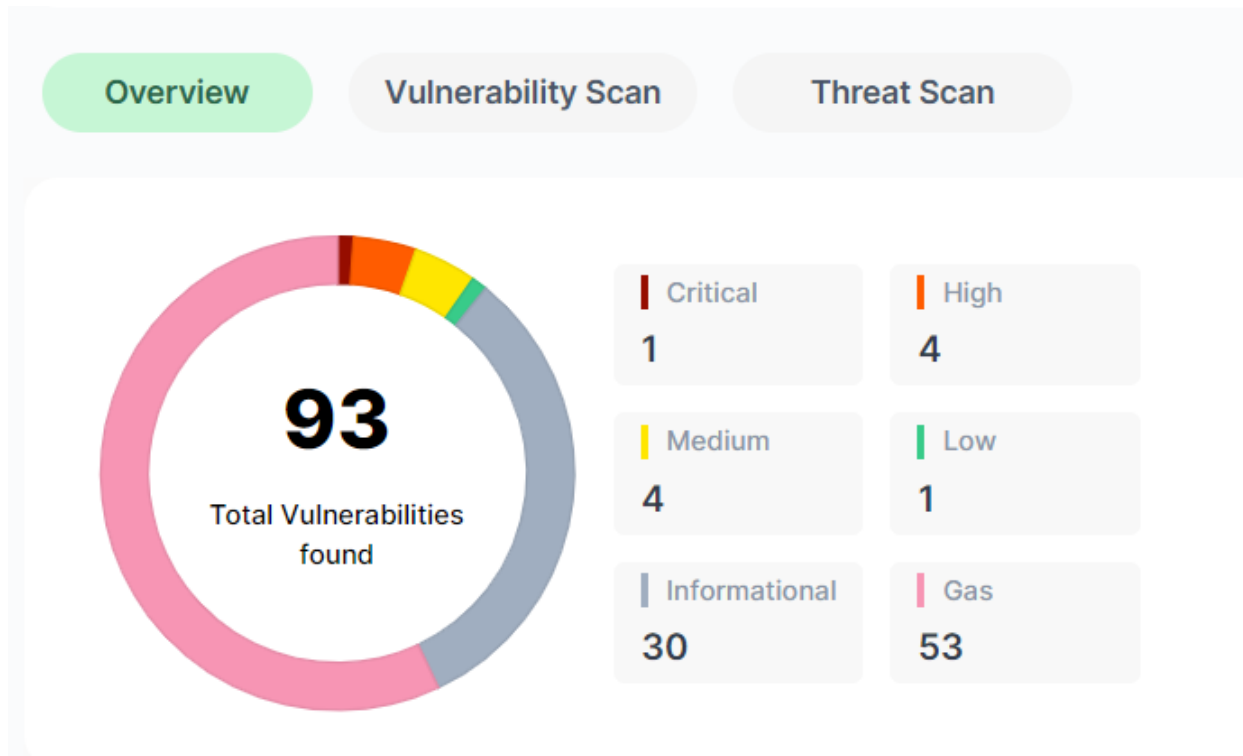
20 Safe Open Zeppelin contracts implementation and
· usage.

pass

21. Fallback function security.

Passed

Security Issues



🔴 Critical Security Issues

Issue # 1:

INCORRECT ACCESS CONTROL

Access control plays an important role in the segregation of privileges in smart contracts and other applications. If this is misconfigured or not properly validated on sensitive functions, it may lead to loss of funds, tokens and in some cases compromise of the smart contract.

The contract is importing an access control library but the function is missing the modifier. It refers to situations where a contract allows unauthorized users to execute certain functions or gain access to restricted resources.

```
//
function _mint(address account, uint256 amount) internal virtual {
    require(account != address(0), "ERC20: mint to the zero address");

    _beforeTokenTransfer(address(0), account, amount);

    _totalSupply = _totalSupply.add(amount);
    _balances[account] = _balances[account].add(amount);
    emit Transfer(address(0), account, amount);
}
```

Remediation:

It is recommended to go through the contract and observe the functions that are lacking an access control modifier. If they contain sensitive administrative actions, it is advised to add a suitable modifier to the same



High Security Issues

Issue # 1:

UNCHECKED TRANSFER

Some tokens do not revert the transaction when the transfer or transfer From fails and returns False. Hence we must check the return value after calling the **transfer** or **transfer From** function.

```

function _transfer(
    address from,
    address to,
    uint256 amount
) internal override {
    require(from != address(0), "ERC20: _transfer from the zero address");

    if (amount == 0) {
        super._transfer(from, to, 0);
        return;
    }

    super._transfer(from, to, amount);
}

```

Remediation:

The issue you're describing is common with tokens that do not follow the standard of reverting on failure in the **transfer** or **transferFrom** functions. Instead of throwing an error, they return a false boolean value when a transfer fails. This can lead to potential vulnerabilities if the return value is not checked, as the smart contract may assume the transfer was successful when it wasn't.

Issue # 2:

Approve Front-Running Attack

The method overrides the current allowance regardless of whether the spender already used it or not, so there is no way to increase or decrease allowance by a certain value atomically unless the token owner is a smart contract, not an account. This can be abused by a token receiver when they try to withdraw certain tokens from the sender's account. Meanwhile, if the sender decides to change the amount and sends another **approve** transaction, the receiver can notice this transaction before it's mined and can extract tokens from both transactions, therefore, ending up with tokens from both the transactions. This is a front-running attack affecting the Bep20 **Approve** function.

```

    */
    function approve(address spender, uint256 amount)
        public
        virtual
        override
        returns (bool)
    {
        _approve(_msgSender(), spender, amount);
        return true;
    }
}

```

Remediation:

Use increaseAllowance and decreaseAllowance Functions

Instead of directly setting a new allowance with approve, you can use the functions increaseAllowance and decreaseAllowance to change allowances incrementally. This mitigates the front-running attack by not resetting the allowance directly but incrementing or decrementing it in a safer way.

Example:

```

solidity
Copy code

// These functions increase or decrease allowances incrementally, preventing front-running
function increaseAllowance(address spender, uint256 addedValue) public returns (bool) {
    _approve(_msgSender(), spender, _allowances[_msgSender()][spender].add(addedValue));
    return true;
}

function decreaseAllowance(address spender, uint256 subtractedValue) public returns (bool)
    _approve(
        _msgSender(),
        spender,
        _allowances[_msgSender()][spender].sub(subtractedValue, "ERC20: decreased allowance");
    );
    return true;
}

```



Medium Security Issues

Issue:

PRECISION LOSS DURING DIVISION BY LARGE NUMBERS

In Solidity, when dividing large numbers, precision loss can occur due to limitations in the Ethereum Virtual Machine (EVM). Solidity lacks native support for decimal or fractional numbers, leading to truncation of division results to integers. This can result in inaccuracies or unexpected behaviors, especially when the numerator is not significantly larger than the denominator.

```
function div(int256 a, int256 b) internal pure returns (int256) {  
    // Prevent overflow when dividing MIN_INT256 by -1  
    require(b != -1 || a != MIN_INT256);  
  
    // Solidity already throws when dividing by 0.  
    return a / b;  
}
```


Remediation:

Instead of dividing directly, multiply the numerator by a precision factor before dividing. This maintains precision by ensuring that you work with larger numbers before truncating.

Example:

If you're performing a division, instead of doing:


solidity

 Copy code

```
uint result = a / b; // This can cause precision loss when 'b' is a large number
```

You should do:

solidity

 Copy code

```
uint precision = 1e18; // 10^18 to retain precision
uint result = (a * precision) / b;
```



Low Security Issues

Issue:

USE OWNABLE2STEP

Ownable2Step is safer than **Ownable** for smart contracts because the owner cannot accidentally transfer the ownership to a mistyped address. Rather than directly transferring to the new owner, the transfer only completes when the new owner accepts ownership.

```

contract Ownable is Context {
    address private _owner;

    event OwnershipTransferred(
        address indexed previousOwner,
        address indexed newOwner
    );


    /**
     * @dev Initializes the contract setting the deployer as the initial owner.
     */
    constructor() {
        address msgSender = _msgSender();
        _owner = msgSender;
        emit OwnershipTransferred(address(0), msgSender);
    }
}

```

Remediation:

Example:

solidity

 Copy code

```


// Before (using Ownable):
import "@openzeppelin/contracts/access/Ownable.sol";

contract MyContract is Ownable {
    // contract logic
}

```

After (using Ownable2Step):

solidity

 Copy code

```

import "@openzeppelin/contracts/access/Ownable2Step.sol";

contract MyContract is Ownable2Step {
    // contract logic
}

```


Informative Security Issue

Issue # 1:

MISSING UNDERSCORE IN NAMING VARIABLES

Solidity style guide suggests using underscores as the prefix for non-external functions and state variables (private or internal) but the contract was not found to be following the same.

```
library SafeMathInt {
    int256 private constant MIN_INT256 = int256(1) << 255;
    int256 private constant MAX_INT256 = ~(int256(1) << 255);

    /**
     * @dev Multiplies two int256 variables and fails on overflow
```

Remediation # 1:

Example:

Example Before Remediation:

solidity Copy code

```
// SPDX-License-Identifier: MIT
pragma solidity ^0.8.0;

contract MyContract {
    uint256 internal data;
    uint256 private secret;

    function _internalFunction() internal {
        data += 1;
    }

    function _privateFunction() private {
        secret += 1;
    }

    function publicFunction() external {
        _internalFunction();
        _privateFunction();
    }
}
```

Remediated Code with Underscore Prefix:

solidity Copy code

```
// SPDX-License-Identifier: MIT
pragma solidity ^0.8.0;

contract MyContract {
    uint256 internal _data; // Remediated: Added underscore to internal variable
    uint256 private _secret; // Remediated: Added underscore to private variable

    function _internalFunction() internal {
        _data += 1; // Updated to use the new variable name with underscore
    }

    function _privateFunction() private {
        _secret += 1; // Updated to use the new variable name with underscore
    }

    function publicFunction() external {
        _internalFunction();
        _privateFunction();
    }
}
```

Issue # 2:

NAME MAPPING PARAMETERS

After Solidity 0.8.18, a feature was introduced to name mapping parameters. This helps in defining a purpose for each mapping and makes the code more descriptive. The following affected files were found to be using floating pragma:

```
contract ERC20 is Context, IERC20, IERC20Metadata {
    using SafeMath for uint256;

    mapping(address => uint256) private _balances;


    mapping(address => mapping(address => uint256)) private _allowances;
```

- **Remediation # 2:**

Identify all mappings in the affected contract.
Assign meaningful names to the mapping parameters for better readability.

Example Before Remediation

solidity

 Copy code

```
// SPDX-License-Identifier: MIT
pragma solidity ^0.8.0;

contract MyContract {
    mapping(address => uint256) balances;
    mapping(uint256 => string) tokenURIs;


    function updateBalance(address user, uint256 amount) public {
        balances[user] = amount;
    }

    function setTokenURI(uint256 tokenId, string memory uri) public {
        tokenURIs[tokenId] = uri;
    }
}
```

Remediation: Using Named Mapping Parameters

With Solidity 0.8.18, we can add names to the mapping parameters to make them descriptive:

solidity

 Copy code

```
// SPDX-License-Identifier: MIT
pragma solidity ^0.8.18;

contract MyContract {
    mapping(address user => uint256 balance) public balances; // Named mapping parameters
    mapping(uint256 tokenId => string uri) public tokenURIs;    // Named mapping parameters

    function updateBalance(address user, uint256 amount) public {
        balances[user] = amount;
    }

    function setTokenURI(uint256 tokenId, string memory uri) public {
        tokenURIs[tokenId] = uri;
    }
}
```

Issue # 3

REDUNDANT STATEMENTS

The contract contains redundant statements where types or identifiers are declared but not used, leading to no action being performed with them. These statements do not generate any code and can be safely removed to clean up the contract.

- **Remediation # 3:**

To remediate the issue of redundant statements, where types or identifiers are declared but not used in a smart contract, you should remove those unused declarations to clean up the contract and improve its readability and efficiency

By removing redundant statements, unused variables, and functions, you improve the clarity, maintainability, and efficiency of the contract. This cleanup ensures the contract is streamlined and follows Solidity best practices.

Issue # 4:

HARD-CODED ADDRESS DETECTED

The contract contains an unknown hard-coded address. This address might be used for some malicious activity. Please check the hard-coded address and its usage.

These hard-coded addresses may be used everywhere throughout the code to define states and interact with the functions and external calls.

Therefore, it is extremely crucial to ensure the correctness of these token contracts as they define various important aspects of the protocol operation.

Remediation # 4:

The presence of **hard-coded addresses** in a smart contract can be a significant risk, especially if the address is not verified or properly validated. Hard-coding addresses can lead to unintended consequences, such as pointing to malicious addresses, incorrect external contracts, or loss of control over key functions. It's crucial to ensure that these addresses are correct, verified, and ideally configurable to avoid compromising the contract's security.

Hard-coded addresses should be avoided in smart contracts due to the risks of misconfiguration, security vulnerabilities, or malicious activity. By making addresses configurable and adding validation and access control mechanisms, you can significantly improve the security and flexibility of the contract, ensuring that key addresses can be updated safely when necessary.

GAS Security Issues

1)

USE OF SAFEMATH LIBRARY

`SafeMath` library is found to be used in the contract. This increases gas consumption than traditional methods and validations if done manually.

Also, Solidity `0.8.0` includes checked arithmetic operations by default, and this renders `SafeMath` unnecessary.

```
contract ACI is ERC20, Ownable {  
    using SafeMath for uint256;  
  
    constructor() ERC20("Acossi Coin", "ACI") {  
  
        /*  
         * _mint is an internal function in ERC20.sol that is c  
         * and CANNOT be called ever again  
         */  
    }  
}
```

Remediation # 1:

We do not recommend using `SafeMath` library for all arithmetic operations. It is good practice to use explicit checks where it is really needed and to avoid extra checks where overflow/underflow is impossible.
The compiler should be upgraded to Solidity version `0.8.0+` which automatically checks for overflows and underflows.

2)

CHEAPER INEQUALITIES IN REQUIRE()

The contract was found to be performing comparisons using inequalities inside the `require` statement. When inside the `require` statements, non-strict inequalities (`>=`, `<=`) are usually costlier than strict equalities (`>`, `<`).

```
404      * - Addition cannot overflow.
405      */
406      function add(uint256 a, uint256 b) internal pure returns (uint256) {
407          uint256 c = a + b;
408          require(c >= a, "SafeMath: addition overflow");
409
410          return c;
411      }
412
413      /**
```


Remediation # 2:

It is recommended to go through the code logic, and, if possible, modify the non-strict inequalities with the strict ones to save `~3` gas as long as the logic of the code is not affected.

3)

LONG REQUIRE/REVERT STRINGS

The `require()` and `revert()` functions take an input string to show errors if the validation fails. This strings inside these functions that are longer than `32 bytes` require at least one additional `MSTORE`, along with additional overhead for computing memory offset, and other parameters.

```
289         address sender,  
290         address recipient,  
291         uint256 amount  
292     ) internal virtual {  
293         require(sender != address(0), "ERC20: transfer from the zero address");  
294         require(recipient != address(0), "ERC20: transfer to the zero address");  
295  
296         _beforeTokenTransfer(sender, recipient, amount);  
297     }
```

Remediation # 3:

It is recommended to short the strings passed inside `require()` and `revert()` to fit under `32 bytes`. This will decrease the gas usage at the time of deployment and at runtime when the validation condition is met.

Conclusion

Smart contracts contain High severity issues! Liquidity pair contract's security is not checked due to out of scope.

Liquidity locking details NOT provided by the team.

Itish note:

Please check the disclaimer above and note, the audit makes no statements or warranties on business model, investment attractiveness or code sustainability. The report is provided for the only contract mentioned in the report and does not include any other potential contracts deployed by Owner.

